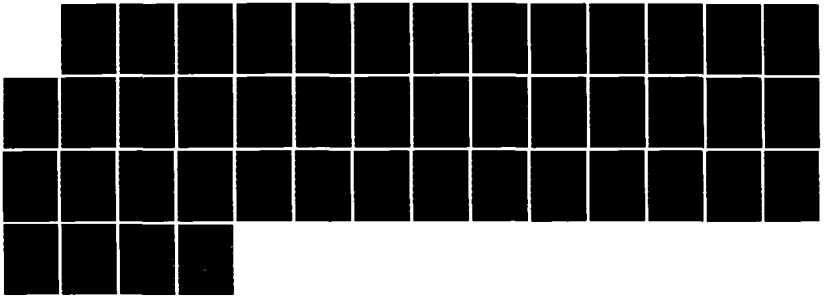
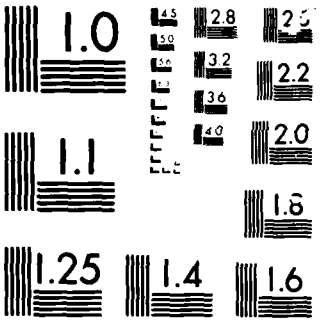
AD-A167 441 PARALLEL COMPLEXITY OF LOGICAL QUERY PROGRAMS(U) 1/1
STANFORD UNIV CA DEPT OF COMPUTER SCIENCE
J D ULLMAN ET AL. 20 DEC 85 STAN-CS-85-1089
UNCLASSIFIED N00014-85-C-0731 F/G 9/2 NL





MICROCOPY CHART

December 1985

Report No. STAN-CS-85-1089

(2)

Parallel Complexity of Logical Query Programs

by

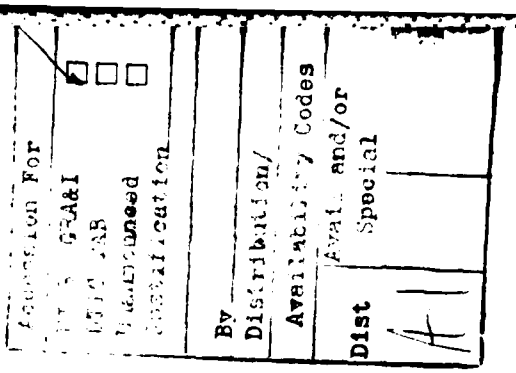
Jeffrey D. Ullman

Allen Van Gelder

Contract No. N00014-85-C-0731

Department of Computer Science

Stanford University
Stanford, CA 94305



Parallel Complexity of Logical Query Programs *

Jeffrey D. Ullman and Allen Van Gelder
Stanford University

December 20, 1985

Abstract

We consider the parallel time complexity of logic programs without function symbols, called logical query programs. We give a PRAM algorithm for computing the minimum model of a logical query program, and show that for programs with the "polynomial fringe property," this algorithm runs in logarithmic time. As a result, the "linear" and "piecewise linear" classes of logic programs are in NC. Then we examine several nonlinear cases in which the program has a single recursive rule that is an "elementary chain." Among such "elementary single rule" programs that are nonlinear, some can easily be shown to have an equivalent linear program, hence are in NC. We show that certain nonlinear programs are related to GSM mappings of a balanced parentheses language, and that this relationship implies the "polynomial fringe property;" hence such programs are in NC. Finally, we describe an approach for demonstrating that certain logical query programs are log space complete for P, and apply it to both elementary single rule programs and nonelementary programs.

1 Introduction

We consider the parallel time complexity of logic programs without function symbols, called logical query programs. Essentially, such programs add the power of a least fix-point operator to relational algebra. With the development of efficient implementations of Prolog and the possibility of massive parallel computation through advances in VLSI techniques, there has been increased interest in the power of this extension. It is well known that the least fix-point operator is a genuine extension to the expressive power of the monotonic subset of relational algebra that excludes set difference [AU79], and that queries expressible in this extended language (i.e., queries representable by function-free Horn clause query programs) are computable in time that is polynomial in the size of the database [CH82]. The parallel complexity of such programs was largely unknown [Var85].

*Supported by NSF grant IST-84-12791, a grant of IBM Corp., and ONR contract N00014-85-C-0731.

Some of these logical query programs involve a thinly disguised transitive closure, which is known to be in $\mathcal{NC}$. Prominent among these are programs employing *linear recursion*. A program is said to have linear recursion when no rule has more than one recursive subgoal.

The classification of programs employing "inherent" nonlinear recursion, i.e., those that cannot be imitated by a program with linear recursion, has been an unresolved question. We shed some light on the nonlinear recursion question in this paper. We present new PRAM algorithms for, and define the "polynomial fringe property" for logical query programs. In our first main result, the Polynomial Fringe Theorem, we show that logical query programs with this property are in $\mathcal{NC}$, hence permit very fast parallel computation, given enough resources. Our second result, the GSM Mapping Theorem, is that certain programs with "inherent" nonlinear recursion have the polynomial fringe property; hence such programs are in $\mathcal{NC}$.

The remainder of the paper is organized as follows: in Sections 2 and 3 we give basic definitions and define the Basic Theorem Problem, by which we measure the complexity of a logical query program. In Section 4 we review the semantics of Horn logic programs, minimum models, and derivation trees. In Sections 5 and 6 we present new PRAM algorithms and prove our first main result, the Polynomial Fringe Theorem. In Section 7 we mention some easy consequences of the Polynomial Fringe Theorem, some of which were already known. In Section 8 we prove our second main result, the GSM Mapping Theorem. In Section 9 we show that slight changes to programs that are in $\mathcal{NC}$ produce programs that are log space complete for $\mathcal{P}$. We conclude in Section 10.

2 Basic Definitions

2.1 Logic Programs

An *atom* is an atomic formula, i.e., a predicate symbol with terms as arguments, as customarily defined in logic. A *literal* is a *polarized* atom, i.e. a positive or negative atom. A *clause* is a disjunction of literals. A *unit clause* has exactly one literal. A *definite clause* has exactly one positive literal. A *negative clause* has no positive literal; the empty clause is considered negative. Strictly speaking, a *Horn clause* is either a definite clause or a negative clause; however, the term is often used loosely as a synonym for "definite clause" when no confusion is likely.

Definite clauses are also called *rules*, and we shall normally use this more descriptive term. The positive literal is then called the *head* of the rule, and the negative literals are called its *subgoals*. A *Horn logic program* is a set of rules. An entity (atom, rule, or logic program) is *function free* if its terms are only variables and constants, i.e., no functors of positive arity appear. An entity is *ground* if its terms contain no variables. For our purposes, a *logical query program* is a function-free Horn logic program; we often abbreviate this to "logic program", or even "program," as that is the only kind we consider.

Definition 2.1: A *basic logic program* is a finite set of definite rules containing two classes of predicate symbols:

- *IDB* (Intentional Database) predicates are those that appear in rule heads, and possibly in subgoals also; $p, p_1, p_2, \ldots$ denote IDB predicates.
- *EDB* (Extensional Database) predicates are those that appear only in subgoals; $q, q_1, q_2, \ldots$ denote EDB predicates.

An *EDB fact* is a positive unit clause over an EDB predicate, with constants as arguments. The constants are drawn from some countable set. An *EDB instance* (or EDB, for short) is a finite set of EDB facts.

An *extended logic program* is the union of a basic logic program and an EDB instance; the predicates appearing in the EDB instance are a subset of the (syntactically defined) EDB predicates in the basic logic program.

Borrowing from the terminology of context free languages, we define a *terminal rule* to be one in which all subgoals are EDB predicates, and a *nonterminal rule* to be one with at least one IDB subgoal.
□

We generally follow the Prolog convention that symbols beginning with an uppercase letter are logic variables, while predicate symbols and constants begin with a lowercase letter.

We shall assume for simplicity that the rules of the basic logic program contain no constants, i.e., all predicate arguments in a rule are variables. Also, we assume that every rule has at least one subgoal. (It is easy to meet these requirements by creating a limited number of additional EDB predicates and requiring certain EDB facts to be present.)

2.2 Dependence Structure and Rank of Rules

The predicates of a basic logic program have a natural partition into *strong components*, based on the following *dependence graph*. Let each IDB predicate symbol p_i and each EDB predicate symbol q_j be a node in the graph; put an arc from p_j (resp. q_j) to p_i if there is a rule whose head is p_i and which has a p_j (resp. q_j) subgoal. The strong component of an IDB predicate consists of just the predicates in its strong component in the dependence graph. Of course, each EDB predicate is a singleton strong component.

Definition 2.2: A (nonterminal) rule is called *recursive* if it has a subgoal in the same strong component as the head. In a recursive rule, a *recursive subgoal* is a subgoal in the same strong component as the head of the rule. Other subgoals are *nonrecursive*. Thus the same atom may be a recursive subgoal in one rule and a nonrecursive subgoal in another.
□

Associated with the dependence graph is the *reduced dependence graph*. Each node in the reduced dependence graph corresponds to a strong component in the dependence graph. Put in an arc from strong component s_1 to s_2 whenever some predicate in s_1 is a subgoal for a rule whose head is in s_2. The reduced graph is acyclic, of course.

Definition 2.3: We define the *rank* of a predicate to be the height (length of longest path to a leaf) of its strong component in the reduced dependence graph; the rank of a rule is the rank of its head. The rank of each EDB predicate is 0. □

We observe that in any rule the rank of a recursive subgoal equals the rank of the head of the rule, while the rank of a nonrecursive subgoal is less than the rank of the head.

3 The Basic Theorem Problem

We shall consider the parallel time complexity of a certain decision problem involving logic programs, which we call the *basic theorem problem* for logic programs. For the basic theorem problem, a basic logic program $\mathbf{P}_I$ is given and regarded as fixed. Assume its IDB predicates are $p_1, \ldots, p_s$ and its EDB predicates are $q_1, \ldots, q_m$. The rules contain only variables, but the input language has a countable set of constants. Recall our convention that a *basic* logic program contains no EDB facts; we use the subscript I as a reminder of this convention. For definiteness here, we denote vectors of constants that occur as arguments by $\bar{a}$, $\bar{b}$, $\bar{c}$, etc. Later, we shall drop the overbar, and let context determine whether a, b, etc. represent single constants or vectors of constants. The input to a problem instance is a ground clause Q of the form

$$p_i(\bar{c}_0) :- q_{j_1}(\bar{c}_1), \ldots, q_{j_N}(\bar{c}_N) \tag{1}$$

and the question is, "Is Q a theorem of $\mathbf{P}_I$?" In other words, the basic theorem problem for $\mathbf{P}_I$ consists of recognizing the set of theorems of $\mathbf{P}_I$ in the form of Eq. 1.

An alternative and equivalent formulation of a problem instance is to define $\mathbf{P}_E$ to be the set of EDB facts, $\{q_{j_1}(\bar{c}_1), \ldots, q_{j_N}(\bar{c}_N)\}$, from the right hand side of Eq. 1, and define $\mathbf{P} = \mathbf{P}_I \cup \mathbf{P}_E$; then ask the question, "Is $p_i(\bar{c}_0)$ a theorem of $\mathbf{P}$?" In both cases we define the *size* of the input to be N, the number of EDB literals, or facts, in the input.

We wish to characterize basic logic programs according to whether their basic theorem problem is in $\mathcal{NC}$, or is *log space complete for* $\mathcal{P}$ ($\mathcal{P}$-complete for short). That the problem is always in $\mathcal{P}$ can be seen from the *Naive Evaluation* algorithm described below (see [CH82] for a complete proof). Loosely speaking, we say a basic logic program is in $\mathcal{NC}$, or is $\mathcal{P}$-complete, when we really mean that its basic theorem problem is. This problem has been called the *data complexity problem* in [CH82,Var82].

Our computational model is a PRAM in which concurrent writes must be consistent (the *common write* model). We are not concerned with representation issues, so make the

convenient assumption that constants in $\mathbf{P}_E$ (i.e., the Herbrand universe of the extended logic program) consist of a reasonably dense subset of some prefix of the natural numbers, so that there is no problem in associating processors and global memory locations with a vector of EDB constants. The treatment of $\mathbf{P}_I$ as fixed is important in our accounting. In particular, there are integers a and b that represent the maximum arities of any IDB predicate and any EDB predicate, respectively, but depend only on $\mathbf{P}_I$ and not on the EDB, $\mathbf{P}_E$, of a particular problem instance. We are assured of having at most Nb different constants in a $\mathbf{P}_E$ of size N. Let v possibly depend on $\mathbf{P}_I$, but not on $\mathbf{P}_E$. The number of v-tuples of EDB constants is then at most $(Nb)^v$, i.e., is polynomial in the size of the input, and we can consider assigning a separate processor or memory location to each such tuple. We assume that once the EDB is given, any processor can check whether a ground atom is an EDB fact and can associate a memory address with a v-tuple of EDB constants in constant time.

4 Semantics of Logic Programs and Derivation Trees

Following [VEK76,AVE82], we define the Herbrand base U of an extended logic program $\mathbf{P}$ to be the set of variable-free atoms containing no predicates or constants other than those occurring in $\mathbf{P}$. An *interpretation* is a subset of U.

With a logic program $\mathbf{P}$ we associate a function T_P from interpretations to interpretations. Let I be an interpretation. We define a ground atom a to be in $T_P(I)$ if and only if there exists in $\mathbf{P}$ a rule $b_0 :- b_1, \ldots, b_n$ and there exists a substitution θ such that:

- a is syntactically identical to $b_0\theta$, the instantiated head of the rule, and
- I contains every instantiated subgoal, $b_1\theta, \ldots, b_n\theta$.

We define a lattice whose set is the powerset of the Herbrand base, and whose partial order is set inclusion. If $\mathbf{P}$ is a definite logic program, then T_P is monotone with respect to this order. By the Knaster-Tarski Fixpoint Theorem every monotone function has a least fixpoint. It is well known [VEK76] that the least fixpoint of T_P is the minimum Herbrand model of $\mathbf{P}$ (i.e., is a subset of every Herbrand model of $\mathbf{P}$). Moreover, in the function-free case there is an integer K such that $T_P^K(\emptyset)$ equals the minimum model. Let us call an interpretation a *partial model* if it is a subset of the minimum model. Then $T_P^k(\emptyset)$ is clearly a partial model for all natural numbers, k.

The EDB portion of the minimum model is simply the set of EDB facts (since by definition EDB predicates never appear in the head of a rule), so we take for granted that EDB facts are included in all partial models we compute. Consequently, when we talk about representing or computing "the model," we mean the IDB portion of the minimum Herbrand model.

Definition 4.1: Given an extended logic program $\mathbf{P}$, a *derivation tree* for a ground atom p^0 is a rooted tree with atoms as nodes and edges between parents and children, such that

- p^0 is the root
- for every internal node r^0 whose children are $r^1, \ldots, r^k$, there is some ground rule instance

$$r^0 :- r^1, \ldots, r^k$$

, where r^i are atoms with various predicate symbols.

- every node is in the minimum model of $\mathbf{P}$; leaves are not necessarily in the EDB.

A *nonterminal derivation tree* is one in which each internal node has at least one IDB atom among its children; i.e., only nonterminal rules were used in the derivation. A *complete derivation tree* is one in which all leaves are EDB facts.

By *path* in a derivation tree we mean a directed path away from the root. The *fringe* of a tree is the set of its leaves. A *tight* derivation tree is one in which no node has an ancestor identical to itself; i.e., no atom occurs twice on one path. □

5 PRAM Algorithms for Logical Query Programs

We begin by describing fairly general parallel algorithms for computing the minimum model of an extended logic program. Analysis of these algorithms leads to a sufficient condition for a basic logic program to be in $\mathcal{NC}$. (Clearly the basic theorem problem of $\mathbf{P}_I$ is in $\mathcal{NC}$ if and only if computation of the minimum model of $\mathbf{P}_I \cup \mathbf{P}_E$ is in $\mathcal{NC}$: Since the Herbrand base is of polynomial size, we can consider all candidates, $p_i(\bar{c})$, in parallel.)

Let N be the length of an input to the basic theorem problem. Recall that the input consists of one IDB ground atom (the "top-level goal") and N EDB ground atoms (EDB facts), and that the problem is to decide whether the EDB facts together with the nonterminal rules imply the top-level goal. The number of constants in the input is $O(N)$.

This problem is well-known to be in $\mathcal{P}$ in view of the algorithm *Naive Evaluation*, which simply computes $T_P^1(\emptyset), \ldots, T_P^K(\emptyset)$ until a fixpoint is reached [AU79,CH82]. If parts (b) and (c) are removed from our Basic Evaluation algorithm below, the result is a PRAM implementation of Naive Evaluation. Each iteration can be done in constant time, but it is easy to find $\mathcal{NC}$ examples, such as "transitive closure," for which $K = O(N)$ for Naive Evaluation. Parts (b) and (c) provide the speed-up needed to bring a large class of programs down to poly-log PRAM time.

Algorithm 5.1: (Basic Evaluation) The constants appearing in the EDB comprise the Herbrand universe. "Instantiations" refer to instantiations of variables to these constants. We shall build the minimum Herbrand model of the extended logic program by constructing a directed graph in which the nodes are the IDB atoms in the Herbrand base, plus a node

for **true**. We call this graph the *implication graph* of the extended logic program. We regard an atom as being in the partial model (and call that atom *true*) whenever there is an arc from **true** to that atom in the implication graph. Initially the implication graph has no edges; the partial model is empty.

Initialization: For each terminal rule, evaluate all its instantiations in parallel; for those proved by the EDB facts, put in an arc from **true** to the node corresponding to the instantiated head of the rule thereby adding it to the partial model. For each other rule, consider all its instantiations in parallel; those such that all EDB subgoals are EDB facts are "live," and are simplified by removing the EDB subgoals. Rule instances with an EDB subgoal for which there is not a matching EDB fact can be discarded from further consideration. If v is the maximum number of variables in any nonterminal rule, then $O(N^v)$ processors can accomplish the foregoing in a constant number of steps.

Iteration: Perform a new iteration stage as long as any new IDB atom was derived (i.e., a *new* arc from **true** to some node was added) in the previous stage.

Part (a) Begin by checking each live rule instance in parallel. Remove any of its remaining (IDB) subgoals that are now "true," i.e., in the partial model. If it has no subgoals left, put an arc from **true** to the head of the rule instance into the implication graph.

Part (b) If a live rule instance has one subgoal left, suppose it has been reduced to $p^1 :- p^2$. Put the arc $p^2 \rightarrow p^1$ into the implication graph; i.e., put in an arc from the node corresponding to the instantiated subgoal to the node corresponding to the instantiated head of the rule.

Part (c) Now transitively close the current implication graph. If a is the maximum number of arguments in any IDB predicate, the implication graph has $O(N^a)$ nodes, so the transitive closure can be done in poly-log time with "only" $O(N^{3a})$ processors. Note that any nodes that become reachable from **true** are now in the partial model. This completes one stage of the iteration.

Termination: When nothing has been added to the partial model in an iteration stage, halt. Lemma 5.1 shows that the minimum Herbrand model has been constructed. For the basic theorem decision problem, the top-level goal atom is a theorem precisely if it is in this model. □

Lemma 5.1: Algorithm 5.1 correctly computes the minimum Herbrand model of the extended logic program.

Proof: The proof follows the proofs in [VEK76,AU79,CH82] that $T_P^k(\emptyset)$ converges to the least fixpoint, finitely in the function-free case. The only difference is that arcs in the implication graph are used also. But arc $a \rightarrow b$ is easily seen to correspond to a theorem (or lemma, if you will), "a implies b," and transitive closure (part (c)) amounts to reasoning by *modus ponens*. ▮

The PRAM time for an iteration stage of Algorithm 5.1 is easily seen to be constant for parts (a) and (b), and poly-log for part (c). In order to show that a basic logic program is in $\mathcal{NC}$, it is sufficient to show that Algorithm 5.1 requires only poly-log stages when applied to this program. Theorem 6.2 provides a general tool for this purpose. But first, we present a modification of Basic Evaluation.

Algorithm 5.2: (Fast Evaluation) All parts of this algorithm are the same as Algorithm 5.1 (Basic Evaluation) except part (c) of the iteration stage, which is replaced by:

Part (c) Now do one "transitive closure step" in the current implication graph. Specifically, in parallel, if $p^3 \rightarrow p^2$ and $p^2 \rightarrow p^1$ are already in the implication graph, then put in arc $p^3 \rightarrow p^1$. Again, if a is the maximum number of arguments in any IDB predicate, the implication graph has $O(N^a)$ nodes, so this step can be done in constant time with "only" $O(N^{3a})$ processors. Note that any nodes that become reachable from true are now in the partial model. □

A little thought reveals that Fast Evaluation computes the same implication graph as Basic Evaluation, due to the monotonicity of both algorithms. Assuming Basic Evaluation does "transitive closure" by repeated application of part (c) of Fast Evaluation, it is further evident that Fast Evaluation is at least as fast as Basic Evaluation. In fact, in the cases we can analyze, both algorithms require $O(\log N)$ stages, but Basic Evaluation may require $O(\log N)$ per stage, while Fast Evaluation requires only constant time per stage. Thus the name "Fast Evaluation" is justified.

6 Polynomial Fringes and $\mathcal{NC}$

In this section we define a key property for basic logic programs, the *polynomial fringe property*, and show that programs with this property run in poly-log time under the Basic and Fast Evaluation algorithms presented in Section 5. Ruzzo has obtained interesting related results for alternating Turing machines with *polynomial sized computation trees* [Ruz80]; however, his tree transformations appear to be based on a different principle from ours, involving a centroid concept. Our methods are more analogous to a method of parallel evaluation of tree expressions used, e.g., in [VSBR83], [MR85], and [AH85]. What is somewhat novel in our use of tree transformations is that our algorithm does not explicitly process (or preprocess) the tree; we only use the tree as a device to *analyze* the algorithm.

Definition 6.1: A basic logic program has the *polynomial fringe property* (relative to a class of EDB's $\mathcal{D}$) if for every EDB (in class $\mathcal{D}$) and every atom in the minimum model of the resulting extended logic program, that atom has a derivation tree whose fringe is of polynomial length in the size of the EDB. If the class $\mathcal{D}$ is not mentioned, it is taken to be all possible EDB's. □

Next we quote a useful lemma, which allows us to restrict attention to tight derivation trees, which have polynomial depth. Recall that a derivation tree is *tight* if no atom occurs twice on one path from the root. Thus in tight derivation trees, a polynomial number of nodes is equivalent to a polynomial fringe length.

Lemma 6.1: Every atom in the minimum model of a logical query program has a tight derivation tree, whose depth is polynomial in the size of the EDB.

Proof: If a complete derivation tree for atom a is not tight, then there is another complete derivation tree for a with fewer nodes and no more leaves, obtained as follows: Find the node b' with identical ancestor b, and substitute the subtree rooted at b' in place of the b subtree. It follows that the *minimum* complete derivation tree for a is tight. Since there are only polynomially many instances of IDB literals in the Herbrand base, and no instance appears twice on one path in a tight derivation, the lemma follows. ∎

The following theorem is a weak form of the Polynomial Fringe Theorem (Theorem 6.3), but serves as an introduction to many of the ideas in a simpler form.

Theorem 6.2: A basic logic program with the polynomial fringe property is in $\mathcal{NC}$.

Proof: We show that for any IDB atom p^0 in the minimum model of $\mathbf{P}$, if p^0 has a complete derivation tree with fringe F, then Algorithm 5.1 requires at most $\log_2 |F| + 1$ stages to put p^0 into the model it constructs, where $|F|$ is the number of atoms in F.

In order to track the progress of the computation as it relates to p^0, we define the *remaining derivation tree* for p^0 to be the original (complete) derivation tree with all nodes that so far have been put into the partial model removed. Accordingly, at the beginning of the first iteration stage, all EDB facts have been pruned from the remaining derivation tree, and it contains only IDB atoms. At the beginning of any iteration stage, the fringe consists of atoms that will be put into the partial model during this stage, as all the required subgoal instances have been put in during preceding stages. Thus every atom in the fringe at the beginning of the stage is pruned from the remaining derivation tree during part (a) of that stage.

The key observation is that any atom that is in the fringe at the *end* of an iteration stage had at least two descendant atoms that were in the fringe at the *beginning* of that stage. For suppose some atom p^1 has only one descendant p^2 in the fringe at the beginning of the stage. Then it must have only one remaining subgoal (i.e., not yet in the partial model). Moreover, every descendant of p^1 other than p^2 must have had only one remaining subgoal. Then arcs existed in the implication graph that form a path from p^2 to p^1, at the beginning of the stage. Since p^2 was in the fringe, the arc from **true** to p^2 was put in during part (a), completing a path from **true** to p^1 in the implication graph. Therefore, during part (c) an arc from **true** to p^1 was put in the implication graph, making p^1 part of the partial model. Thus p^1 with only one fringe descendant cannot be in the new fringe.

Since every new fringe atom had two fringe descendants in the remaining derivation tree of the preceding stage, the size of the fringe halves during each iteration stage. Thus, the root is put into the partial model within $\log_2 |F| + 1$ stages. Since each stage requires poly-log time and uses a polynomial number of processors, the theorem follows. ∎

Example 6.1: Suppose atom p^1 has the remaining derivation tree shown in Fig. 1 after the initialization stage of the Basic Evaluation algorithm, and assume it has no other derivations. It is evident that Naive Evaluation requires four steps to derive p^1. Let us

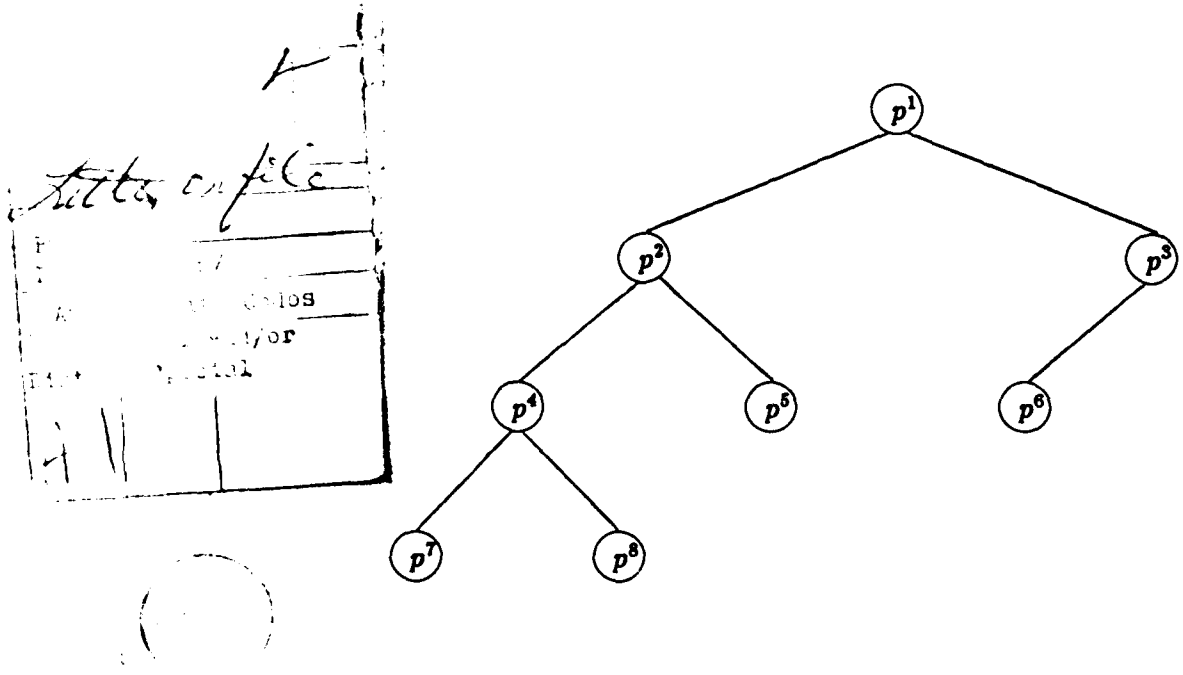


Figure 1: An unbalanced remaining derivation tree

track the progress of Basic Evaluation.

(0) Implication graph empty. Remaining derivation tree is as shown in Fig. 1, with four fringe nodes.

(1a) Implication graph gets arcs from **true** to p^7, p^8, p^5, and p^6.

(1b) Implication graph gets arc $p^6 \rightarrow p^3$.

(1c) Transitive closure puts in arc from **true** to p^3. Remaining derivation tree is reduced to p^1, p^2, and p^4.

(2a) Implication graph gets arc from **true** to p^4.

(2b) Implication graph gets arcs $p^4 \rightarrow p^2$ and $p^2 \rightarrow p^1$.

(2c) Transitive closure puts in arcs from **true** to p^2 and p^1, as well as $p^4 \rightarrow p^1$. Remaining derivation tree is now empty.

Thus Basic Evaluation derives p^1 in two stages. Observe that stage (2c) requires two "transitive closure steps." (Realistically, a third step is also needed to see that nothing new is found.) Because Fast Evaluation does not do a full transitive closure in each stage, it requires three stages. On problems of any nontrivial size, the reduced time per stage more than makes up for the additional stages. □

Programs with the polynomial fringe property run in $\log^2 N$ PRAM time under Basic Evaluation. It is straightforward to simulate this computation with a $\log^3 N$ depth Boolean circuit of bounded fan-in gates. (It is easy to see that $\log^k N$ PRAM time in our algorithms translates to $\log^{k+1} N$ depth for a bounded fan-in circuit, and to $\log^k N$ depth for polynomial fan-in.) The class $\mathcal{NC}^k$ essentially consists of problems that can be solved by "uniform" circuits of depth $O(\log^k N)$ using a polynomial number of bounded fan-in gates; we omit the technical uniformity condition [Coo84]. Since our class of problems includes "transitive closure," $\log^2 N$ depth is the best known bound and is likely optimal, so it is of some interest to show that Fast Evaluation achieves this bound on programs with the polynomial fringe property. Our proof uses the node-counting method of [MR85].

Theorem 6.3: (Polynomial Fringe Theorem) A basic logic program with the polynomial fringe property is in $\mathcal{NC}^2$.

Proof: We show that for any IDB atom p^0 in the minimum model of **P**, if p^0 has a nonterminal complete tight derivation tree T with $|T|$ nodes, then Algorithm 5.2 (Fast Evaluation) requires at most $\log_{\frac{4}{3}} |T|$ stages to put p^0 into the model it constructs. Since the polynomial fringe property, together with Lemma 6.1, guarantee that $|T|$ is polynomial in N, the theorem follows.

In order to track the progress of the computation as it relates to p^0, we define an *effective derivation tree* for p^0. As in Theorem 6.2, this tree is initially a complete derivation tree; however the updating operation is more complicated. As before, all nodes that are put into the partial model (as well as nodes corresponding to EDB facts) are removed. In addition, all "chains" are shortened in accordance with arcs in the current implication graph. Specifically, starting with the root of the current effective derivation tree and working down, if there is a sequence of nodes $p^1, p^2, \ldots, p^r$ such that:

- p^i has only one child, p^{i+1} for $1 \leq i < r$;
- p^r has no children or several children;
- All arcs $p^{i+1} \rightarrow p^i$ for $1 \leq i < r$ are in the current implication graph;

then we call $p^1, \ldots, p^r$ a *maximal chain*. Given a maximal chain $p^1, \ldots, p^r$, let i be the largest index such that $p^i \rightarrow p^1$ is in the current implication graph. Then we update the effective derivation tree by making p^i the (only) child of p^1, and removing $p^2, \ldots, p^{i-1}$ from the effective derivation tree. This type of updating continues until it is no longer possible, always operating on the maximal chain closest to the root when there is a choice.

Thus edges (e.g., between p^1 and p^i in the above example) can appear in the effective derivation tree that were not in the original derivation tree.

Let T_k be the effective derivation tree after stage k, and let F_k be its fringe. Accordingly, in T_0, all EDB facts have been pruned from T, and it coincides with the remaining derivation tree at this point. However, as the iteration proceeds, effective derivation trees diverge from remaining derivation trees, in general.

At the beginning of any iteration stage, the fringe consists of atoms that will be put into the partial model during this stage, as all the required subgoal instances have been put in during preceding stages. Thus every atom in the fringe at the beginning of the stage is pruned from the effective derivation tree during part (a) of that stage. No change occurs during part (b).

In order to describe the changes in the effective derivation tree that occur during part (c) of stage k of Fast Evaluation, we define a homeomorphism h on T_k, the effective derivation tree at the beginning of stage k (i.e., before part (a)). Let $U_k = h(T_k)$ be formed by mapping every node of T_k that has exactly one child into its nearest descendant that either is a leaf or has at least two children. Other nodes of T_k map to themselves, and provide the "names" of the nodes in U_k. In particular, F_k maps to itself. For $u \in U_k$, or equivalently, for $u \in T_k$ such that $h(u) = u$, define $h^{-1}(u)$ to be the set of nodes that map into u, and define $chain(u) = h^{-1}(u) - \{u\}$. Every internal node of U_k has at least two children. It follows (*cf.* complete binary trees) that $|U_k| < 2|F_k|$.

For any $u \in U_k$, order $chain(u)$ nodes by distance from the root, and call them $p^1, \ldots, p^r$. During part (c) of stage $k+1$ of the Fast Evaluation algorithm, arcs $p^3 \rightarrow p^1$, $p^4 \rightarrow p^2$, $p^5 \rightarrow p^3$, etc., will be put into the implication graph. The effective derivation tree will be updated as follows: p^3 will become the child of p^1 and p^2 will be removed; p^5 will become the child of p^3 and p^4 will be removed; etc. [1] In summary, if $chain(u)$ contains r nodes in T_k, it will shrink to $\lceil \frac{1}{2} r \rceil$ nodes in T_{k+1}. We have the following relationships:

$$
\begin{aligned}
|T_k| &= \sum_{u \in U_k} |h^{-1}(u)| = |U_k| + \sum_{u \in U_k} |chain(u)| \\
|T_{k+1}| &= |U_k| - |F_k| + \sum_{u \in U_k} \left\lceil \tfrac{1}{2} |chain(u)| \right\rceil \\
|U_k| &< 2|F_k|
\end{aligned}
$$

[1] Arcs $p^4 \rightarrow p^2$ and others remain in the implication graph, of course, but play no further role in this effective derivation tree.

If we regard $|T_k|$ as fixed and regard $|U_k|$, $|F_k|$, and $|chain(u)|$ as variables, and look for the maximum value of $|T_{k+1}|$, it can be shown (with Lagrange multipliers, e.g.) that the maximum occurs when $|chain(u)| = 1$ for all $u \in U_k$. With this substitution, $|U_k| = \frac{1}{2}|T_k|$ and

$$|T_{k+1}| = 2|U_k| - |F_k| < \tfrac{3}{4}|T_k|$$

Thus $|T_k|$ will be 0 after at most $\log_{\frac{4}{3}} |T|$ stages.

To complete the proof, it is only necessary to note that whenever a node is removed from the effective derivation tree, either it is in the partial model (i.e., it was in the fringe), or it has a parent. Thus the root p^0 must be in the partial model when the effective derivation tree has been reduced to nothing. ∎

7 Preliminary Results on Programs in $\mathcal{NC}$

In this section we mention several known results that are easy corollaries of the Polynomial Fringe Theorem.

7.1 Context Free Language Recognition

Deciding whether an input string is in a (fixed) context free language, i.e., the *nonuniform word problem*, has been shown to be in $\mathcal{NC}$ in [Ruz80] through a series of simulations. In [VSBR83] it is transformed into a polynomial evaluation problem. Here we obtain the result as a corollary, with a straightforward algorithm as well.

Corollary 7.1: The word problem for a fixed context free language is in $\mathcal{NC}^2$.

Proof: Let G be a grammar for the language, in Chomsky normal form (other epsilon-free forms will also work). The nonterminal symbols in the grammar (denoted somewhat unconventionally by lowercase letters) will correspond to IDB predicates, and the terminal symbols (denoted by Greek letters) will correspond to EDB predicates. For each production $a \rightarrow bc$ we create a nonterminal rule

$$a(I, K) :\!- b(I, J), c(J, K).$$

and of course $a \rightarrow \alpha$ becomes the terminal rule

$$a(I, J) :\!- \alpha(I, J).$$

Now our Fast Evaluation algorithm based on this IDB will not necessarily run in poly-log time on an arbritrary EDB. However, to recognize strings in the language, we only need to process certain EDB's. In particular, if the input string is $\alpha_1 \cdots \alpha_n$, we put the facts $\alpha_1(0, 1)$, $\alpha_2(1, 2)$, ..., $\alpha_n(n-1, n)$ into the EDB. We take all such EDB's to be our

class S. It is evident that an IDB atom $a(i,j)$ is derivable only if it has a derivation tree with fringe equal to the substring of symbols $i+1$ through j of the input string. (Note that this fact depends on having no epsilon productions.) This fringe is of length $j-i$. Hence the program has the polynomial fringe property relative to S and, restricted to EDB's in S, is in $\mathcal{NC}$. ∎

7.2 Linear and Piecewise Linear Programs

That linear logic programs are in $\mathcal{NC}$ has independently been discovered by M. Vardi and D. Maier, P. Kanellakis, and possibly others. (We are not aware of any published version of this "folk theorem.") Piecewise linear programs are a natural generalization. Perhaps they comprise the largest previously known class of $\mathcal{NC}$ logic programs.

Definition 7.1: A *piecewise linear* logic program is one in which each rule has at most one recursive subgoal. If in addition each rule contains at most one IDB subgoal, the program is said to be *linear*. If a rule contains two or more recursive subgoals, then that rule and the logic program are said to be *nonlinear*. Recall that any nonrecursive IDB subgoal is necessarily of lower rank than the head of the rule. □

Corollary 7.2: Any piecewise linear logical query program is in $\mathcal{NC}$.

Proof: We use induction on k, the rank of derivable atoms, and show that atoms of rank k have a polynomial bound Q_k on the fringe length of their minimum complete derivation trees.

The basis, $k=0$, is trivial as these are EDB predicates, and we may choose $Q_0 = 1$.

Now for $k > 0$ assume that derivable atoms of rank less than k have complete derivation trees whose fringe lengths are bounded by polynomial Q_{k-1}. Consider the rules of (any) one strong component of rank k. Any derivable atom in that component is derivable with these rules, together with rules of lower rank. For these predicates (of rank k), each internal node of a complete derivation tree has at most one recursive child and a number of nonrecursive children bounded by s, the maximum number of subgoals in any rule of the IDB. All the nonrecursive children have rank less than k, so have fringes bounded by Q_{k-1}. Thus the total fringe length is within a factor of sQ_{k-1} of the depth. By Lemma 6.1, every derivable atom has a polynomial depth derivation tree. Thus there is a polynomial bound Q_k, on the fringe length of minimum complete derivation trees for atoms of rank k. Since the maximum rank is bounded, the program has the polynomial fringe property. ∎

7.3 Elementary Chain Rules

When we consider the class of nonlinear basic logic programs, even those with a single recursive rule, and therefore a single recursive predicate p, the water turns muddy very

quickly. However, if we further restrict ourselves to elementary chain rules (defined below), we can state some results. In this section we define terms and state results for certain nonlinear programs that have linear equivalents. In subsequent sections we examine the more difficult question of nonlinear programs with no linear equivalents.

Definition 7.2: We say that a logic program is in *join normal form* if:

- Every variable that appears in the head of a recursive rule also appears in some subgoal of that rule.
- Every variable in a recursive rule appears in some recursive atom (either the head or a recursive subgoal).

□

Definition 7.3: Two basic logic programs are said to be *equivalent with respect to a set of IDB predicates* S if the minimum models of both programs extended with the same EDB, restricted to predicates in S, are the same. □

Any basic logic program can be rewritten into one in join normal form that is equivalent with respect to the original IDB predicates. If it started with one recursive rule it will finish with one recursive rule. In general, new predicates and nonrecursive rules for them will be introduced.

Example 7.1: Suppose variables X and Y appear only in nonrecursive subgoals q_1, q_2 and p_2 of a recursive rule in a basic logic program, which we suppose to be:

$$p_1(U,W,Z) :- q_1(X,U), q_2(X,Y), p_2(U,V,Y), p_1(V,W,Z).$$

and suppose $p_1, \ldots, p_4$ are the IDB predicates already defined. We can define a new nonrecursive IDB predicate p_5 with the rule:

$$p_5(U,V) :- q_1(X,U), q_2(X,Y), p_2(U,V,Y).$$

Observe that the offending variables X and Y are "projected out" in relational algebra terminology. The variables U and V appearing in p_5 are just those that appeared in q_1, q_2 or p_2 and in a p_1 atom as well. Now we can replace the rule given for p_1 by:

$$p_1(U,W,Z) :- p_5(U,V), p_1(V,W,Z).$$

□

Definition 7.4: An *elementary chain* is (1) an ordered list of subgoals (atoms) all of whose arguments are variables and (2) an ordered partition of each predicate's arguments, for each predicate appearing in the list (possibly in several subgoals). Each partition is into left and right blocks, such that:

- Certain variables appear only in the left block of the first subgoal, and are called the *left block* of variables of the chain.
- Certain variables appear only in the right block of the last subgoal, and are called the *right block* of variables of the chain.
- All variables that do not appear in the left or right blocks appear precisely in two adjacent subgoals. The variables in the right block of a subgoal (other than the last subgoal) appear in the same order in the left block of the following subgoal.

An *elementary chain rule* is a rule whose head contains predicate p, in which the arguments of p can be partitioned into two blocks, called the *left block* and *right block*, and the subgoals can be ordered to form an elementary chain, and moreover:

- The variables in the left block of the head of the rule appear in the same order in the left block of the chain of subgoals.
- The variables in the right block of the head of the rule appear in the same order in the right block of the chain of subgoals.
- If p appears as a subgoal, the p subgoals' block definitions are the same as the head's block definitions.

An *elementary strong component* is one for which there is a simultaneous assignment of left and right blocks to all predicates in that component that makes all rules for those predicates elementary chain rules. An *elementary program* is one that contains only elementary components and is in join normal form.

Finally, an *instantiated elementary chain* or *instantiated elementary chain rule* is one in which the variables have been instantiated to constants. □

Clearly, we can permute the arguments of p to make the left block precede the right block, so we assume this is the case. We write $p(X, Y)$ letting X represent the vector of variables in the left block and Y the right block. We can represent chains succinctly by writing their arguments above and between their predicate symbols, like this:

$$\begin{array}{cccccccccc} U & & V & & W & & X & & Y & & Z \\ & q_0 & & p & & q_1 & & q_0 & & q_2 & \end{array}$$

Here the leftmost subgoal is $q_0(U, V)$ and it is followed by $p(V, W)$, etc. The left block of the whole chain is U and the right block is Z.

If the blocks are understood, we can omit the variables without ambiguity when writing out uninstantiated chains. It is also apparent that we can replace a p subgoal by a chain of subgoals with fresh variables and appropriate unifications, as illustrated in the next example; this gives a correct "top-down expansion" of the rule. In other words, elementary strong components can be manipulated much like context-free grammars,

treating predicates in the component as nonterminals and predicates of lower rank as terminals. We shall usually restrict attention to simple cases, in which there is only one nontrivial strong component and all other predicates are EDB predicates.

Example 7.2: The following is an elementary chain rule:

$$p(X,Y,Z) :- q_1(X,U), p(U,V,W), q_2(V,W,Y,Z).$$

The left block of p consists of its first argument, and its right block consists of the second and third arguments. Thus it could be represented first as:

$$\begin{array}{cccccccc} X & & YZ & & X & U & VW & YZ \\ & p & & :- & & q_1 & p & & q_2 \end{array}$$

and then even more succinctly as $P \rightarrow q_1 P q_2$, adopting the notation of productions in a context free grammar. We associate the predicate symbol p with the nonterminal symbol P in the grammar. If we make a new copy by subscripting all variables with 1, then unify its head with $p(U,V,W)$ and "substitute," we obtain an expanded elementary chain rule:

$$\begin{array}{cccccccccc} X & & YZ & & X & U & U_1 & V_1W_1 & VW & YZ \\ & p & & :- & & q_1 & q_1 & p & q_2 & q_2 \end{array}$$

which obviously corresponds to the derivation $P \stackrel{*}{\Rightarrow} q_1 q_1 P q_2 q_2$. □

We now consider several cases of nonlinear programs with an elementary chain rule as the only recursive rule. These cases are in join normal form, have one IDB predicate p, and have several EDB predicates, q_0, q_1, q_2, etc. Although we can write them and think about them as binary predicates, recall that each variable symbol actually represents a vector of variables. We organize the cases by the number of subgoals in the recursive rule.

The first case has two subgoals.

$$\begin{array}{ll} \mathbf{P_1} & p(X,Y) :- p(X,U), p(U,Y). \\ & p(X,Y) :- q_0(X,Y). \end{array}$$

This is a version of transitive closure, well known to be in $\mathcal{NC}$. Note that the left and right blocks must have the same arity in cases like this one, where there are consecutive p subgoals.

The next two cases have three subgoals.

$$\begin{array}{ll} \mathbf{P_2} & p(X,Y) :- p(X,U), p(U,V), p(V,Y). \\ & p(X,Y) :- q_0(X,Y). \end{array}$$

$$\begin{array}{ll} \mathbf{P_3} & p(X,Y) :- p(X,U), q_1(U,V), p(V,Y). \\ & p(X,Y) :- q_0(X,Y). \end{array}$$

To see that they are both in $\mathcal{NC}$, we characterize their complete derivation trees, and show that they have the polynomial fringe property. We associate the grammars G_2 with $\mathbf{P_2}$ and G_3 with $\mathbf{P_3}$, where G_2 consists of $P \rightarrow PPP \mid q_0$ and where G_3 consists of $P \rightarrow Pq_1P \mid q_0$.

Theorem 7.3: The Basic Theorem problems of $\mathbf{P}_1$, $\mathbf{P}_2$ and $\mathbf{P}_3$ are in $\mathcal{NC}$.

Proof: Because the rules are elementary chain rules, it is sufficient to characterize the fringes of the complete derivation trees, which are instantiated elementary chains. It is sufficient to prove the theorem for $\mathbf{P}_2$ and $\mathbf{P}_3$, as $\mathbf{P}_1$ can be simulated by $\mathbf{P}_3$ with q_1 set to the diagonal relation consisting of a tuple $q_1(c,c)$ for every EDB constant, c.

An easy induction on the number of rule applications shows that the $\mathbf{P}_2$ fringes are odd length chains of q_0's, that the $\mathbf{P}_3$ fringes are chains of the form $(q_0q_1)^*q_0$, and that all such chains of EDB atoms are the fringes of valid complete derivation trees. Now suppose some fringe (of either $\mathbf{P}_2$ or $\mathbf{P}_3$) contains two occurrences of q_0 in an odd-numbered position that both have the same instantiations of their left block arguments; i.e., there is a $q_0(a,b_1)$ and somewhere to its right there is a $q_0(a,b_2)$ in the fringe. Then by removing $q_0(a,b_1)$ and the atoms to its right, up to but not including $q_0(a,b_2)$, we obtain a shorter chain that is a valid fringe of a derivation tree for the same root atom. Thus the original derivation tree was not minimum. Therefore, for every atom in the minimum model, there is a derivation tree for that atom whose fringe does not have such a repetition. Since the left block of q_0 can only be instantiated in polynomially many ways, it follows that $\mathbf{P}_2$ and $\mathbf{P}_3$ have the polynomial fringe property. ∎

The above proof, although not difficult, introduces a paradigm for showing that a program has the polynomial fringe property:

(1) Characterize the fringes that the rules of the program can produce with the help of the associated grammar.

(2) Show a polynomial bound such that if the number of occurrences in the fringe of an instantiated atom exceeds that bound, then that fringe is not minimum.

(3) Conclude that every derivable atom has a derivation whose fringe length is polynomially bounded.

This paradigm is developed further in the next section.

The programs examined so far are only "weakly nonlinear" in a certain sense. Using induction on the number of rule applications, it is easy to show that the linear programs:

$\mathbf{P}_1'$
$$p(X,Y) :- q_0(X,U), p(U,Y).$$
$$p(X,Y) :- q_0(X,Y).$$

$\mathbf{P}_2'$
$$p(X,Y) :- q_0(X,U), q_0(U,V), p(V,Y).$$
$$p(X,Y) :- q_0(X,Y).$$

$\mathbf{P}_3'$
$$p(X,Y) :- q_0(X,U), q_1(U,V), p(V,Y).$$
$$p(X,Y) :- q_0(X,Y).$$

are equivalent to $\mathbf{P}_1$, $\mathbf{P}_2$ and $\mathbf{P}_3$, respectively, in the sense of Definition 7.3. This equivalence also follows from Lemma 8.6.

8 Nonlinear Programs in $\mathcal{NC}$

We now turn to programs that have no linear equivalent. When these programs have a close analogy with a "balanced parentheses" language, we can draw upon that analogy to characterize their derivation trees, and show that they have the polynomial fringe property. We "implement" the analogy by means of GSM mappings.

Definition 8.1: Informally, a *generalized sequential machine* (*GSM*) is a nondeterministic finite automaton that emits an "output string" in addition to consuming an input symbol on each "move." The output string is normally over a different alphabet from the input. It is convenient to think of a GSM in terms of a graph in which the nodes are states and the arcs are labeled "a/γ" where a denotes an input symbol (consumed, not ϵ) and γ denotes an output string (emitted, possibly empty). There is one start state and a nonempty set of accepting states. If α is an input string to GSM M and β is the concatenation of the output strings emitted by some accepting computation of M on α, we call β a *mapped element* of α. The *GSM mapping* of input string α, written $M(\alpha)$ is the set of mapped elements of α. A GSM is *ϵ-free* if each arc is labeled with a nonempty output string. For formal details, see [HU79]. □

We now introduce some terminology for discussing strings associated with a context free grammar. Let G be a context free grammar with nonterminal alphabet V and terminal alphabet T, and let L be the language generated by G.

Definition 8.2: A *symbol* is an element of $V \cup T$; a *string* is a sequence of symbols. A *letter* is a terminal symbol, i.e., an element of T; a *word* is a sequence of letters. Frequently we use a superscript, such as a^1, to designate a particular occurrence of symbol a. The "1" superscript does not mean the first symbol of the string, however. A *nonterminal production* is one with a nonterminal on the right side; a *terminal production* is one with no nonterminals on the right side. A *sentential form* of G is a string produced from the start symbol by zero or more productions. A *nonterminal* sentential form of G is one that uses only nonterminal productions. □

Definition 8.3: The *Dyck language on one kind of parentheses*, D_1, is the context free language whose grammar is:

$$G_{D_1} \qquad \begin{array}{l} S \rightarrow E \mid \epsilon \\ E \rightarrow EE \mid [E] \mid [] \end{array}$$

Intuitively, this language consists of all "balanced parentheses" words, including the empty word, ϵ. We denote the language consisting of D_1 concatenated with an end-marker \$ by $D_1\$$. □

It turns out that working with $D_1\$$ avoids many technicalities where GSM mappings are concerned; e.g., we avoid the question of mapping the empty word.

To characterize sentential forms of D_1, we need the notion of "nesting depth." It is convenient to label the "points" between successive symbols of a string with integers. The "point" to the left of the string is 0; the point following the i-th symbol is i; and therefore the point to the right of a string of length n is n.

Definition 8.4: For strings over an alphabet containing [and] (and possibly other symbols) we assign a *left-depth* to each point by starting with 0 at point 0, and moving left to right. We add 1 whenever we cross an [, and subtract 1 whenever we cross a]. Similarly, we assign a *right-depth* by starting with 0 at the point to the right of the string, and moving to the left. Now we add 1 whenever we cross an], and subtract 1 whenever we cross a [. For strings with an equal number of [and] it is apparent that the left-depth equals the right-depth at every point, so we can call the common value the *depth* of that point. We shall restrict attention to such strings, and define the depth of the symbols in the string. We define the depth of a [in a string to be the depth of the point to its left, and define the depth of a] to be the depth of the point to its right. For other symbols, the depth of either adjacent point may be taken. □

Lemma 8.1: A word over $\{[,]\}$ is in D_1 if and only if it has an equal number of [and], and every letter has a nonnegative depth.

Proof: (This well-known fact is easier to re-prove than to find a reference for.) The $\Rightarrow$ direction is trivial. The $\Leftarrow$ direction is shown by induction on k, where $2k$ is the word length. The basis, $k = 0$ or $k = 1$, is immediate. For $k > 1$, assume the $\Leftarrow$ direction for words of length less than $2k$.

In a word w of length $2k$ in which every letter has nonnegative depth, find the *rightmost* [that has maximum depth, and call it $[^2$. The $[^2$ must be followed by a] of the same depth, which we denote by $]^2$. The depth 0 case is easy, so assume the depth is $d > 0$. $[^2]^2$ must be followed by another]. If they are preceded by a [, then let $w = w_1[[^2]^2]w_3$. Clearly $w_1[]w_3$ has all nonnegative depths, so is in D_1 by the inductive hypothesis, from which it follows that w_1Ew_3 is a sentential form of G_{D_1} (there is no other way to get a [] pair). The production $E \rightarrow [E]$ shows that $w_1[E]w_3$ is also a sentential form; hence $w \in D_1$. If $[^2]^2$ is preceded by a], it also has depth d; call it $]^1$. By maximality of d there is a $[^1$ immediately to the left of $]^1$. Let $w = w_1[^1]^1[^2]^2w_3$. The remainder is similar to the previous case, except that the production $E \rightarrow EE$ is used. ∎

Theorem 8.2: (GSM Mapping Theorem) Let **P** be a basic logic program with a distinguished IDB predicate p, other IDB predicates, $p_1, p_2, \ldots$, and several EDB predicates $q_1, q_2, \ldots$, all of which p depends upon. Let **P** have only elementary chain rules. Let G be the context free grammar obtained by considering P as the start symbol, considering

$P, P_1, P_2, \ldots$ as nonterminals, and considering $q_1, q_2, \ldots$ as terminals. The productions of G correspond to the rules of $\mathbf{P}$ in the obvious way. That is, the right side of a production corresponds to the chain of subgoals of a rule, and P, P_1, ... correspond to p, p_1, ..., respectively. Let L be the language generated by G. If L is the GSM mapping of $D_1\$$ for some GSM M, then $\mathbf{P}$ has the polynomial fringe property, hence is in $\mathcal{NC}$.

Proof: Let F be the fringe of a minimum complete derivation tree of $\mathbf{P}$, expressed as an elementary chain, for some atom a in the minimum model. By "minimum" we mean minimum fringe length. Let $\hat{F}$ be the corresponding word obtained by using just the predicate symbols of F. Clearly $\hat{F} \in L$. Let $\alpha \in D_1\$$ be a minimum length word in $M^{-1}(\hat{F})$. That is, $\hat{F} \in M(\alpha)$. We note that $|F| = |\hat{F}| = O(|\alpha|)$.

From now on we consider a particular computation of M on α that produces $\hat{F}$. Let $s_1, \ldots, s_r$ be the sequence of states of M in this computation, excluding the final (accepting) state. We partition $\hat{F}$ into subwords $\{\hat{F}_i \mid \hat{F}_i = M(\alpha_i), i = 1, 2, \ldots, r\}$, where $\alpha_1 \cdots \alpha_r$ are the letters of α. We correspondingly partition F into $F_1, \ldots, F_r$. Let b_i be the left block of F_i, that is, b_i is the vector of EDB constants whose components are the instantiations of variables that occur in both the first subgoal of F_i and the last subgoal of F_{i-1} (or in the head, if $i = 1$). We define an *ID* for this computation of M on α that outputs $\hat{F}$ as the triple $ID(i) = (\alpha_i, s_i, b_i)$, where:

- α_i is the next symbol to be read by M.
- s_i is the state of M just before reading α_i.
- b_i is the left block of F_i.

Furthermore, let d_i denote the depth of α_i in α, as defined in Def. 8.4. The relationships are illustrated in Fig. 2. Note that the depth for each [is associated with the "point" to its left, and the depth for each] is associated with the "point" to its right.

In connection with one computation of M we get a sequence of *ID*'s: $ID(1), \ldots, ID(r)$, and a sequence of nonnegative depths $0 = d_1, \ldots, d_r = 0$.

Claim 1:
For $1 \leq i < j \leq r$, if $d_i = d_j$, then $ID(i) \neq ID(j)$.

Proof of Claim 1:

Suppose there are distinct positions in α, say $i < j$, such that $d_i = d_j$ and $ID(i) = ID(j)$. From the definition of depth and the fact that $\alpha_i = \alpha_j$, it follows that there are an equal number of [and] in the subword $\alpha_i \cdots \alpha_{j-1}$. It follows immediately that $\alpha' \in D_1\$$.

Now we remove the subchain F_i through F_{j-1} from the original fringe F. What remains is still an elementary chain, as now b_j, which equals b_i, connects properly to F_{i-1} (or to the left block of the head, if $i = 1$). Also we remove the subword $\hat{F}_i$ through $\hat{F}_{j-1}$ from F, and we remove $\alpha_i \cdots \alpha_{j-1}$ from α, giving words F' and α'. The situation is depicted in Fig. 3.

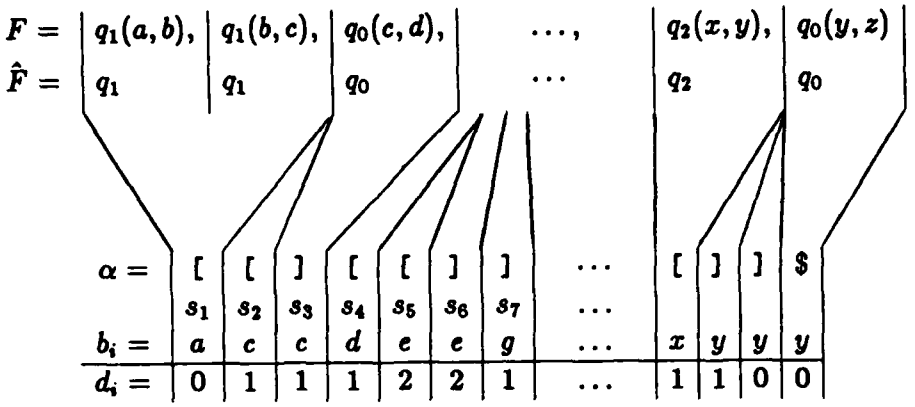


Figure 2: Derivation fringe for $p(a,z)$, *ID* of a GSM computation, and nesting depth.

Since $s_i = s_j$, M was in the same state before reading α_i as before reading α_j. Also, $\alpha_i = \alpha_j = \alpha'_i$. Thus the transition $s_i \rightarrow s_{j+1}$ consuming α_j and emitting $\hat{F}_j$ is legal for M. Define a computation of M on α' that mimics the computation on α except on $\alpha_i \cdots \alpha_{j-1}$. When α'_i is reached, make the transition $s_i \rightarrow s_{j+1}$ consuming α'_i and emitting $\hat{F}'_i$. This computation is legal for M operating on α', so $\hat{F}' \in M(D_1\$)$. Moreover, by the minimality of α for $\hat{F}$, we have that $\hat{F}'$ is a proper subword of $\hat{F}$.

That is, $|F'| < |F|$ and F' is the fringe of a complete derivation tree for the same atom a. This contradicts the assumption that the original derivation tree was minimum. It follows that $ID(i) \neq ID(j)$ for any distinct α_i and α_j at the same depth. □

We next show that the depth of any letter in α is polynomially bounded whenever F corresponds to a minimum derivation. The argument is a more complicated version of the one that showed that *ID*'s at the same depth must be distinct.

Claim 2:
Let d_{max} be the maximum depth of any letter in α. Then d_{max} is polynomially bounded.
Proof of Claim 2:

Let m, $1 \leq m \leq r$ be a position of maximum depth; i.e., α_m has depth d_{max}. Select a subsequence $(i_0, \ldots, i_{d_{max}})$ of $(1, \ldots, m)$ such that:

- d_{i_k}, the depth of α_{i_k}, equals k.
- $d_i > k$ for all i such that $i_k < i \leq m$.

We call this the *left subsequence*. Clearly $ID(i_k)$ is defined for each k, $0 \leq k \leq d_{max}$ ($i_{d_{max}} = m$). Similarly, select a *right subsequence* $(j_0, \ldots, j_{d_{max}})$ of $(r, r-1, \ldots, m)$ such that:

- d_{j_k}, the depth of α_{j_k}, equals k.

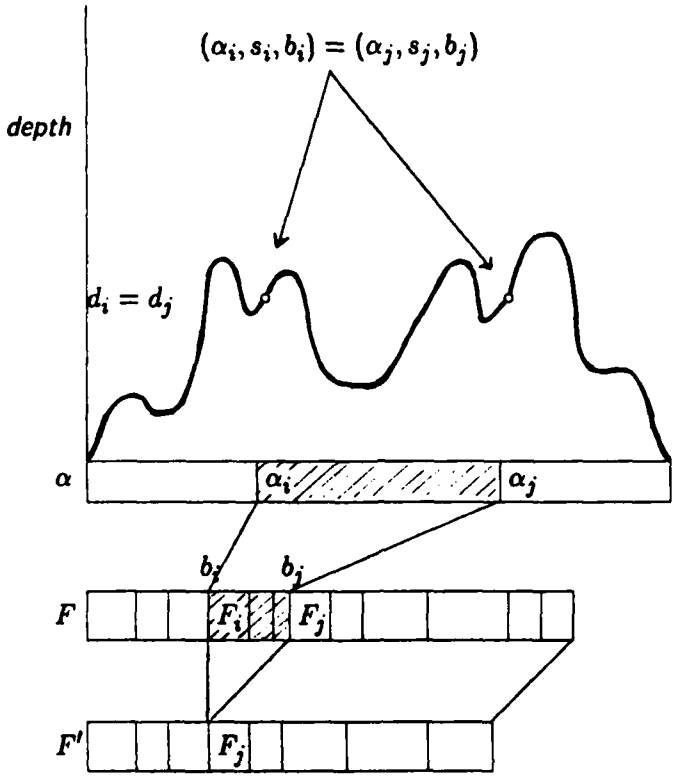


Figure 3: Illustration of Claim 1. IDs cannot repeat at same level.

- $d_j > k$ for all j such that $m \leq j < j_k$.

Again $ID(j_k)$ is defined for each k, $0 \leq k \leq d_{max}$. Finally, for each depth $0 \leq d \leq d_{max}$ we form the pair $Z_d = (ID(i_d), ID(j_d))$.

Suppose there are two distinct integers, $0 \leq d < e \leq d_{max}$, such that $Z_d = Z_e$. Let us simplify subscripting by setting $i = i_d$, $j = i_e$, $k = j_e$, and $l = j_d$. Note that $i \leq j \leq k \leq l$. We now remove *two* subwords of α, namely $\alpha_i \cdots \alpha_{j-1}$ and $\alpha_k \cdots \alpha_{l-1}$, obtaining a shorter word α'. α' clearly has an equal number of ['s and]'s, so "depth" is defined. The main point is that the depth in α of all letters in $\alpha_j \cdots \alpha_{k-1}$ is at least e, so that the depths of corresponding letters in α' are at least d, hence nonnegative. Thus α' is also in $D_1\$$. We also define $\hat{F}'$ and F' by removing subwords and subchains corresponding to the letters removed from α. The situation is depicted in Fig. 4.

It is straightforward that M maps α' into $\hat{F}'$ with an accepting computation, and that F' is the fringe of a complete derivation tree that derives the same atom as F derived. The details are similar to the argument for Claim 1. Since α was minimal for F, F' is strictly shorter than F. This contradicts the assumption that F was minimum; therefore Z_d and

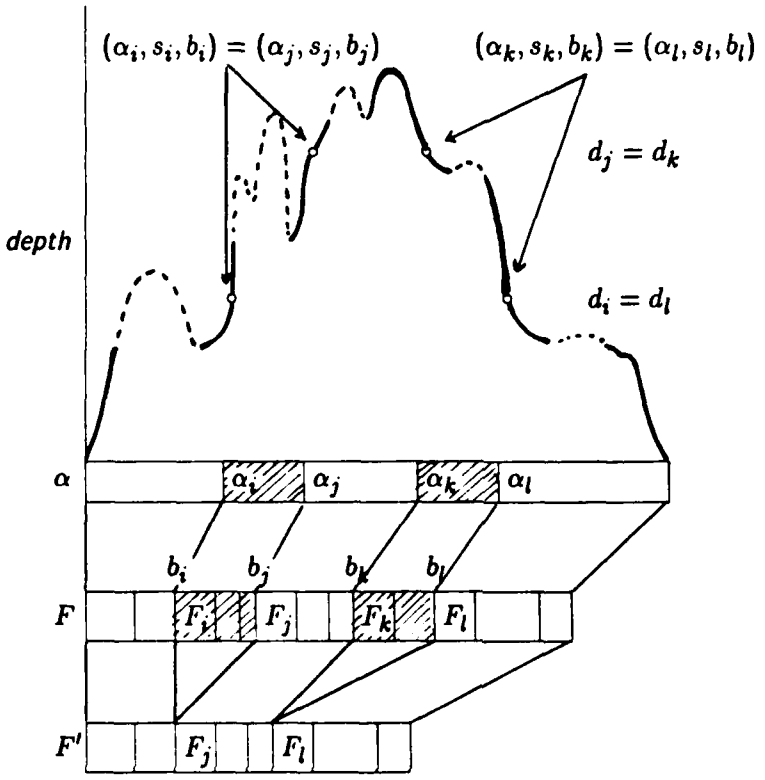


Figure 4: Illustration for Claim 2. Certain *pairs* of IDs cannot repeat. Dotted curve denotes positions not in left or right subsequence.

Z_e must have distinct values whenever $d \neq e$. There are only polynomially many possible values for Z_d, so d_{max} is polynomially bounded. □

To summarize, the depth of α is polynomial, and the same *ID* cannot occur twice at the same depth, thereby bounding the number of symbols of α at each depth, so the length of α is polynomially bounded. Since $|F| = O(|\alpha|)$, the theorem follows. ∎

A *sequential transducer* is essentially a GSM that can also make transitions and produce output without consuming an input symbol; such transitions are called ϵ-*moves*.[2] We remark that this generalization gains us nothing, as sequential transducer mappings of $D_1\$$ give the same family of languages as GSM mappings of $D_1\$$. To see this, suppose that w is a possible output string of sequential transducer T operating on input $\alpha \in D_1\$$, where α is written with ϵ's inserted wherever the sequential transducer will make an ϵ-move in its computation. Form γ by replacing each [in α by [[, replacing each] in α by]], and replacing each ϵ in α by []. A GSM is easily constructed (not dependent on α, of course)

[2]These moves are not to be confused with moves that consume an input symbol, but output nothing.

that imitates T to produce w as a computation on γ.

As an application of the theorem on GSM mappings of $D_1\$$ we consider:

$$\begin{array}{ll} \mathbf{P_4} & p(X,Y) :- q_1(X,U), p(U,V), q_2(V,W), p(W,Y). \\ & p(X,Y) :- q_0(X,Y). \end{array}$$

Let G_4 and L_4 be the grammar and language corresponding to $\mathbf{P_4}$. G_4, given by $P \rightarrow q_1 P q_2 P \mid q_0$. We observe that all words in L_4 have an equal number of q_1's and q_2's, and that for any $k > 0$, words of the form $q_1^k(q_0q_2)^k\ q_1^k(q_0q_2)^k \cdots q_1^k(q_0q_2)^k\ q_0$ are in L_4. It appears that all such words cannot be recognized by any *finite-turn push-down automaton*, as defined in [GS66,HU79], hence are not in a linear context-free language. We therefore do not expect to find an equivalent linear program for $\mathbf{P_4}$, as we did for $\mathbf{P_1}$, $\mathbf{P_2}$, and $\mathbf{P_3}$.

To apply the GSM mapping theorem, we define a GSM M_4 as shown in Fig. 5. The arc label "a/γ" means "consume input symbol a and output string γ.

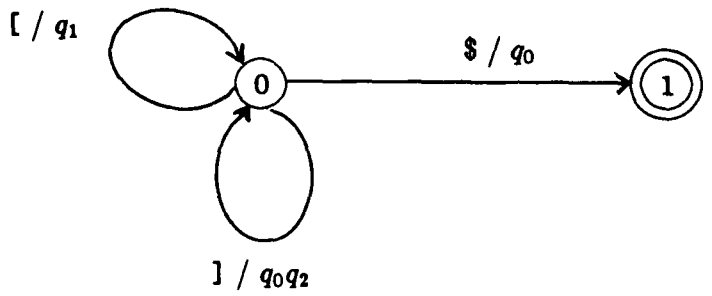


Figure 5: The GSM M_4, with start state 0 and accepting state 1.

Lemma 8.3: A word w of q_0, q_1, and q_2 symbols is in L_4 if and only if $w \in M_4(D_1\$)$.

Proof: ($\Rightarrow$) Proceed by induction on k, the number of applications of the nonterminal production of G_4. Note that $|w| = 3k + 1$. For $k = 0$, $M_4(\$) = \{q_0\}$. For $k > 0$, assume the lemma is true in the $\Rightarrow$ direction for derivations with less than k nonterminal production applications. The first production of a k step derivation of w replaces the starting P symbol, say P^0, by $q_1P^1q_2P^2$. Then P^1 and P^2 have derivations of fewer than k nonterminal steps that yield w^1 and w^2 such that $w = q_1w^1q_2w^2$. By the inductive hypothesis, there exist α and β in $D_1\$$ that are mapped by M_4 into w^1 and w^2. Let α' be α with its \$ end-marker removed and define

$$\gamma = [\alpha']\beta$$

It is clear that $\gamma \in D_1\$$. Now we claim that $M_4(\gamma) = \{q_1 w^1 q_2 w^2\}$. The only change is that M_4 emits the q_0 at the end of w^1 (as part of $q_0 q_2$) when it reads the "new"] of γ instead of when it reads the \$ of α.

($\Leftarrow$) Proceed by induction on the length $2k+1$ of the word γ in $D_1\$$. The base case, $k = 0$, is trivial. For $k > 0$, assume the lemma holds in the $\Leftarrow$ direction for shorter words, which are of the form $2k'+1$ for some $0 \leq k' < k$. In a word γ of length $2k+1$ find the *rightmost* [letter that has depth 0, and call it $[^1$. This letter must either be the first letter of γ, or must be preceded by a] of depth 0. There is precisely one] of depth 0 to the right of $[^1$, and it is immediately followed the the \$ end-marker; call this letter $]^1$.

If $[^1$ is not the first letter of γ, partition γ into $\alpha'\beta$, where β consists of $[^1$ and all following letters. Let α be α' with \$ appended. It is clear that both α and β are in $D_1\$$. Let $M_4(\alpha) = \{w^1\}$ and $M_4(\beta) = \{w^2\}$. (The mappings are singleton sets, since M_4 is deterministic.) By the inductive hypothesis, w^1 and w^2 are in L_4. But then so is $w = q_1 w^1 q_2 w^2$, and again it is easy to see that $M_4(\gamma) = \{w\}$.

If $[^1$ is the first letter of γ, partition γ into $[^1\alpha']^1\$$. By definition of $[^1$, α' (as a subword of γ) contains no letters of depth 0. Consequently, α' (as a word) is in D_1. Let α be α' with \$ appended; $\alpha \in D_1\$$. Let $M_4(\alpha) = \{w^1\}$. By the inductive hypothesis, w^1 is in L_4. But then so is $w = q_1 w^1 q_2 q_0$, and again it is easy to see that $M_4(\gamma) = \{w\}$. ∎

Corollary 8.4: $\mathbf{P}_4$ has the polynomial fringe property, hence is in $\mathcal{NC}$.

∎

Corollary 8.5: The following program is in $\mathcal{NC}$:

$$\begin{array}{ll} \mathbf{P}_{5.0.2} & p(X,Y) \text{ :- } q_1(X,U), p(U,W), p(W,Y). \\ & p(X,Y) \text{ :- } q_0(X,Y). \end{array}$$

Proof: For each EDB constant c add an atom $q_2(c,c)$ to the EDB, then run the algorithm on $\mathbf{P}_4$. ∎

For our next application of the GSM Mapping Theorem, we show that $\mathbf{P}_{5.0.2}$ can be generalized to a family of elementary single rule programs with only one nonrecursive subgoal that are in $\mathcal{NC}$.[3] The next program represents a family of programs indexed by i and j, where $0 \leq i \leq j$, and $j > 0$. The notation $P^{(i)}$ denotes i occurrences of P.

$$\begin{array}{lll} \mathbf{P}_{5.i.j} & p(X,Y) \text{ :- } & p(X,U_1), p(U_1,U_2), \ldots, p(U_i,V), \\ & & q_1(V,W_1), p(W_1,W_2), p(W_2,W_3), \ldots, p(W_j,Y). \\ & p(X,Y) \text{ :- } & q_0(X,Y). \end{array}$$

which is succinctly expressed by the grammar $G_{5.i.j}$ consisting of $P \rightarrow P^{(i)} q_1 P^{(j)} \mid q_0$. The complexity of $\mathbf{P}_{5.i.j}$ in the general case is open. However, we can show that the cases $i = 0$

[3] Pure recursions, $P \rightarrow P \cdots P\ P \mid q_0$, are obviously in $\mathcal{NC}$, being equivalent to $P \rightarrow q_0 \cdots q_0\ P \mid q_0$.

and $i = 1$ are in $\mathcal{NC}$ for all j. The case $i = 1$ relies on the following lemma to show that it is essentially the same as the case $i = 0$. It is immediate that two elementary programs are equivalent in the sense of Definition 7.3 if their corresponding grammars generate the same language.

Lemma 8.6: Let G be a grammar with nonterminal symbol P in which the only productions for P are:

$$P \rightarrow P\gamma P \mid q_0$$

Here q_0 is a terminal and γ is a string possibly containing both terminals and nonterminals. Then L, the language generated by G, is also generated by G', where G' is obtained from G by replacing the productions for P by:

$$P \rightarrow q_0\gamma P \mid q_0$$

Proof: We observe that the derivation in G:

$$P \Rightarrow P\gamma P \Rightarrow P\gamma P\gamma P$$

is ambiguous, in that the third string can be derived by replacing either the first or the last P in $P\gamma P$; consequently, when parsing a string, we can assume that such derivations are always obtained by replacing the last P. It follows that any word in L has a derivation in which the P immediately to the left of γ is always replaced by q_0, and never by $P\gamma P$. In other words L is also generated by the grammar G'. ∎

To apply the GSM mapping theorem in the cases $i = 0$ and $i = 1$, we define $M_{5.i.j}$ as shown in Fig. 6. Observe that $M_{5.i.j}$ uses more of the power of the GSM Mapping Theorem than M_4 in that it accepts only a subset of $D_1\$$, yet it does not use the full power, since it is deterministic.

Lemma 8.7: Let $i = 0$ or $i = 1$. A word w of q_0,and q_1 symbols is in $L_{5.i.j}$ if and only if $w \in M_{5.i.j}(D_1\$)$.

Proof: It is sufficient to prove the case $i = 0$, for then the case $i = 1$ follows by Lemma 8.6.

($\Rightarrow$) Proceed by induction on k, the number of applications of the nonterminal production of $G_{5.0.j}$. Note that k is the number of q_1 symbols in w, and $|w| = (i + j)k + 1$. For $k = 0$, $M_{5.0.j}(\$) = \{q_0\}$. For $k > 0$, assume the lemma is true in the $\Rightarrow$ direction for derivations with less than k nonterminal production applications. Let the rightmost occurrence of q_1 in w be in w_m. Let $P \stackrel{*}{\Rightarrow} w$ be a leftmost derivation, i.e., at each step the leftmost nonterminal symbol is replaced by the right hand side of a production. Then the last use of $P \rightarrow q_1P^{(j)}$ in this leftmost derivation produces the occurrence of q_1 in w_m. There are at least j occurrences of q_0 after w_m. Thus the substring $w_m \cdots w_{m+j}$ contains $q_1q_0^{(j)}$. We define w' by replacing this substring of w by a single q_0. Clearly

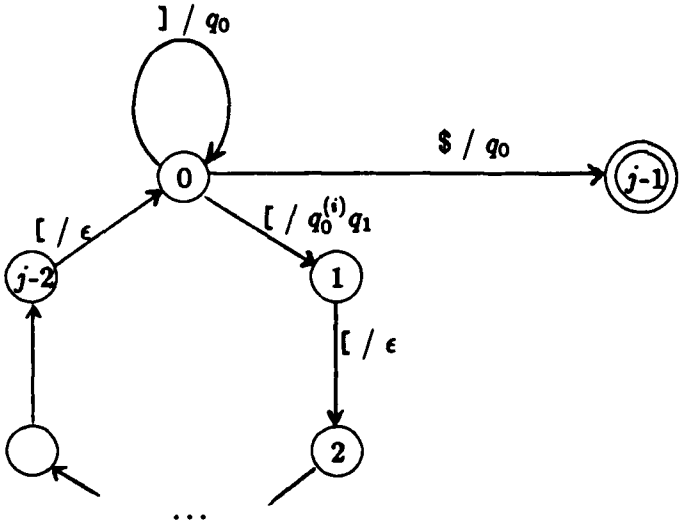


Figure 6: The GSM $M_{5.i.j}$, with start state 0 and accepting state j-1, where $i = 0$ or $i = 1$.

$w' \in L_{5.0.j}$, and the inductive hypothesis applies to show that there is some $\alpha' \in D_1\$$ such that $w' \in M_{5.0.j}(\alpha')$. Partition α' into $\alpha'_1\alpha'_2$ such that $M_{5.0.j}$ emits w'_m in response to the first symbol of α'_2. Thus α'_2 contains only]'s. Now form α by inserting $j-1$ ['s followed by $j-1$]'s between α'_1 and α'_2. Then $w \in M_{5.0.j}(\alpha)$.

($\Leftarrow$) Proceed by induction on the length, $2k(j-1)+1$ of the word α in $D_1\$$. Note that the ['s of α must form k groups of $j-1$ each, or else $M_{5.0.j}$ will not accept α. The base case, $k = 0$, is trivial. For $k > 0$, assume the lemma holds in the $\Leftarrow$ direction for shorter words. In a word α of length $2k(j-1)+1$ find the *rightmost* group of $j-1$ ['s; they must be followed by $j-1$]'s. Remove these $2(j-1)$ letters, creating α'.[4] Clearly $\alpha' \in D_1\$$ and is accepted by $M_{5.0.j}$. Let w' be the string emitted by $M_{5.0.j}$ operating on α', and let w'_m be the symbol emitted in response to the letter immediately after the removed letters. $w' \in L_{5.0.j}$ by the inductive hypothesis. w'_m must be a q_0 because there are no ['s further to the right in α'. In the derivation tree for w' replace the derivation for w'_m, which was $P \Rightarrow q_0$, by the derivation $P \Rightarrow q_1 P^{(j)}$; then replace these new P's by q_0's, giving a new string w. Clearly, $M_{5.0.j}(\alpha) = \{w\} \subseteq L_{5.0.j}$. ∎

Theorem 8.8: Let $i = 0$ or $i = 1$. $\mathbf{P}_{5.i.j}$ has the polynomial fringe property, hence is in $\mathcal{NC}$.

[4] There may be more]'s to the right of the removed letters.

Proof: Immediate from Lemma 8.7 and the GSM Mapping Theorem. ∎

We obtain the "mirror images" $\mathbf{P}_4^R$, $\mathbf{P}_{5.0.j}^R$ and $\mathbf{P}_{5.1.j}^R$ by reversing the subgoal chains in the obvious manner. They are also in $\mathcal{NC}$, therefore.

In Section 9, Definition 9.1, we define a recursive subgoal in an elementary chain rule to be *cut off* if it not at either end of the subgoal chain and is either *partially* or *completely* surrounded by nonrecursive subgoals. Cut-off recursive subgoals play an important part in simulating Boolean circuits for $\mathcal{P}$-completeness proofs, apparently because such subgoals have no variables in common with the head of the rule. In that section we conjecture that any elementary single rule program in which the recursive rule has two cut off recursive subgoals is $\mathcal{P}$-complete.

Recall that elementary programs, being in join normal form, cannot have two consecutive nonrecursive subgoals; consequently, any rule with k nonrecursive subgoals has at least $k-1$ cut off recursive subgoals, and may have more.

Based on our conjecture, and assuming $\mathcal{NC} \neq \mathcal{P}$, the following program appears to be the longest elementary single rule program that is in $\mathcal{NC}$ and whose recursive rule contains two EDB subgoals.

$$\begin{array}{ll} \mathbf{P}_6 & p(X,Y) :- p(X,T), q_1(T,U), p(U,V), q_2(V,W), p(W,Y). \\ & p(X,Y) :- q_0(X,Y). \end{array}$$

Let G_6 and L_6 be the grammar and language corresponding to $\mathbf{P}_6$. Thus G_6 is given by $P \rightarrow Pq_1Pq_2P \mid q_0$. That $\mathbf{P}_6$ is in $\mathcal{NC}$ is immediate from Lemma 8.6 and the fact that $\mathbf{P}_4$ is in $\mathcal{NC}$.

This is about as far as we can go with elementary single rule programs in $\mathcal{NC}$. We shall see in the next section that several ways of extending such rules with more subgoals (remaining in join normal form) result in programs that are $\mathcal{P}$-complete. We conjecture that this is always the case.

However, the GSM Mapping Theorem applies to programs with more than one recursive rule, provided each that rule is an elementary chain rule.

Example 8.1: In this example a single nonelementary chain rule is first transformed into two elementary chain rules, then the theorem is applied.

$$\begin{array}{ll} \mathbf{P}_7 & p(X,Y) :- p(X,V), q_1(V,W), p(Y,W). \\ & p(X,Y) :- q_0(X,Y). \end{array}$$

It is convenient to identify $p^R(Y,X)$ with $p(X,Y)$, $q_0^R(Y,X)$ with $q_0(X,Y)$, and $q_1^R(Y,X)$ with $q_1(X,Y)$. This leads to the elementary chain rules:

$$\begin{array}{ll} \mathbf{P}_7' & p(X,Y) :- p(X,V), q_1(V,W), p^R(W,Y). \\ & p(X,Y) :- q_0(X,Y). \\ & p^R(X,Y) :- p(X,V), q_1^R(V,W), p^R(W,Y). \\ & p^R(X,Y) :- q_0^R(X,Y). \end{array}$$

Let G_7 and L_7 be the grammar and language corresponding to $\mathbf{P}'_7$. Thus G_7 is given by the following productions (P is the start symbol):

$$\begin{aligned} P &\rightarrow Pq_1P^R \mid q_0 \\ P^R &\rightarrow Pq_1^RP^R \mid q_0^R \end{aligned}$$

It is easy to show that L_7 consists of certain odd-length words with q_0 or q_0^R in all the odd positions, and q_1 or q_1^R in the even positions. The first position must contain q_0. Let us interpret every symbol after the first as a left parenthesis if it does not have superscript R and as a right parenthesis if it does have superscript R. The specification of L_7 is completed by requiring that the symbols after the first, thus interpreted, comprise a "balanced parentheses" word. The proof that this characterizes L_7 is a straightforward induction in each direction, and is omitted.

To apply the GSM mapping theorem, we define M_7 as shown in Fig. 7

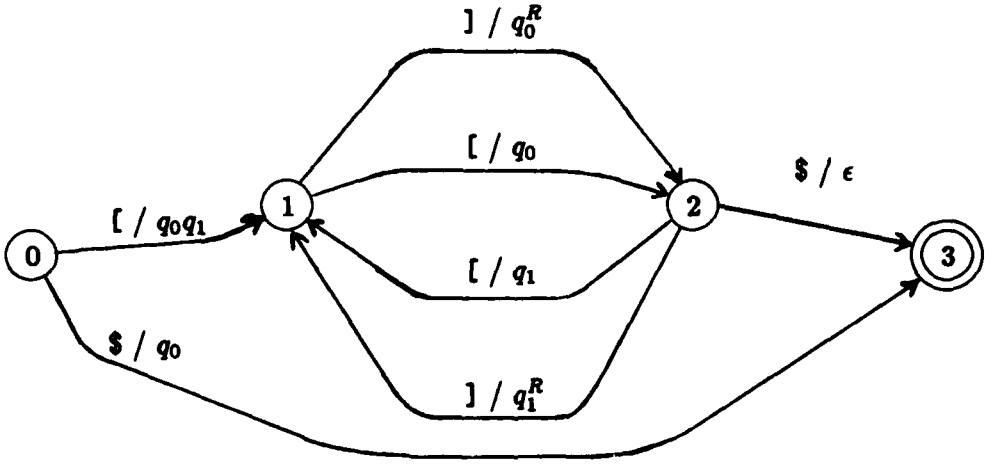


Figure 7: The GSM M_7, with start state 0 and accepting state 3.

Clearly, $M_7(D_1\$)$ implements the characterization of L_7 given above. Consequently (subject to the proof of that characterization), $\mathbf{P}'_7$, and hence $\mathbf{P}_7$, are in $\mathcal{NC}$. □

Unfortunately, the idea of replacing nonelementary chains by a set of elementary chains does not generalize to (truly) tertiary predicates, at least as far as we can see.

Example 8.2: Consider the program:

$\mathbf{P}_8$ $\quad p(X,Y,Z) :- p(X,U,V), q_1(U,V,W), p(Y,W,Z).$
$\quad\quad\quad\quad\quad\quad p(X,Y,Z) :- q_0(X,Y,Z).$

We can define $p_1, \ldots, p_5$ to be p under the various permutations of its arguments, but we cannot maintain the left block as a single argument in the resulting rules. Consequently some "substitutions" are incompatible, and the CFG analogy breaks down. □

9 $\mathcal{P}$-Complete Logical Query Programs

In this section we show $\mathcal{P}$-completeness for a number of logical query programs that are in some sense just beyond the boundaries of the ones we have already considered. Assuming that $\mathcal{NC} \neq \mathcal{P}$, this indicates that some of the $\mathcal{NC}$ "templates" for logical query programs of the type we are considering cannot be further generalized, at least not in obvious ways. Other formulations of logical query programs are possible, of course.

In particular, Cosmadakis and Kanellakis [CK85] have studied formulations with more of a relational database "flavor," for which they coined the name "sirup" (for SIngle RUle Program). For example, they have obtained some results for *typed rules*, which are defined in analogy with typed tableaux. E.g., if U appears in the first argument of one occurrence of p, and appears in the second argument of some other occurrence of p in the same rule, then that rule is not typed.

9.1 $\mathcal{P}$-Complete Elementary Single Rule Programs

Elementary chain rules, as remarked before, have a close analogy with context free grammars. The GSM Mapping Theorem states that GSM mappings of $D_1\$$ are in $\mathcal{NC}$. Yet it is known that *any* CFL can be represented as a GSM mapping of D_2, the Dyck language on 2 kinds of parentheses [Gre73]. It is instructive therefore to demonstrate that elementary chain programs can be $\mathcal{P}$-complete.

We now consider the smallest (in total literal count) $\mathcal{P}$-complete elementary single rule program:

$$\mathbf{P}_{11} \qquad \begin{array}{l} p(X,Y) :- q_1(X,U), p(U,V), p(V,W), q_1(W,Y). \\ p(X,Y) :- q_0(X,Y). \end{array}$$

The proof that $\mathbf{P}_{11}$[5] is $\mathcal{P}$-complete provides a paradigm that can be used for many programs. Observe that the subgoal chain has two occurrences of p that are "cut off" from the end of the chain by q_1 atoms. The basic idea is that when the rule is instantiated, the head "represents" the output of a particular gate of a circuit, and the two recursive subgoals "represent" its inputs. We want to simulate the circuit by setting up an EDB based on the circuit that has this property for each gate g in the circuit: $p(g0, g9)$ is true (derivable) if and only if gate g outputs a 1.

A useful intuition is to interpret $p(g0, g9)$ also as meaning there is an electrical path from $g0$ to $g9$ allowing current to flow. If g is an **and** gate, we want the only possibilty to

[5] Our numbering of programs has some gaps.

be a "series" path that goes through one input, returns to a point associated with g ($g4$ in the case of $\mathbf{P}_{11}$), and then goes through the other input. Thus current can flow through g if and only if it can flow through both of g's inputs. For an **or** gate we want "parallel" paths, so that current can flow through g if and only if it can flow through at least one of its inputs.

We now describe a subclass of monotone Boolean circuits that can be simulated by $\mathbf{P}_{11}$. First, let us designate the class of all monotone Boolean circuits by C, and the subclass of monotone Boolean circuits with only two inputs, which can be designated *left* and *right* arbitrarily, by C_1. It is well known that any circuit in C can be transformed in log space to an equivalent circuit in C_1.

We designate our special subclass by C_2. The binary circuit elements for C_2 are **and** and **or** gates with precisely two distinguished inputs, called the *left input* and *right input*, and one or two distinguished outputs, a *left output*, or a *right output*, or both. In our terminology an output of one gate connects as input to precisely one other gate; "fan out" is achieved by having multiple outputs on a gate. In addition, there are source elements with no inputs and one or two outputs that emit logical 0s or 1s, called **0** and **1** gates appropriately. Their outputs are also distinguished as *left* or *right*. Finally, for a circuit to be in C_2, we require that only left outputs be connected to left inputs and only right outputs be connected to right inputs.

Lemma 9.1: *There is a log space transformation from C to C_2 that preserves the output values of circuits.*

Proof: Suppose some element g in a C_1 circuit has outputs that are connected to $m > 1$ left inputs. Let the left and right inputs to g be e_l and e_r, respectively. Transform the circuit by introducing an additional **and** gate g' and an additional **1** gate g_1. Let gate h be one of the gates connected to an output of g. Connect $m - 1$ outputs from g' to all of the elements that g was connected to, except h, and remove those g connections; designate the g output that goes to h as the *left* output of g, and give g a right output that connects to the right input of g'. Finally, connect the left output of g_1 to the left input of g'. It is evident that this produces an equivalent circuit, and that the transformation can be repeatly carried out in log space until no gate has outputs connected to more than one left input. A corresponding transformation can remove outputs connected to multiple right inputs. ∎

Therefore, we are justified in calling a basic logic program P-complete if we can use it to simulate circuits in C_2. More precisely, we need a log space transformation from C_2 circuits into EDB relations and a particular "theorem" to decide, such that the theorem holds in the extended logic program if and only if the circuit outputs a 1.

We now demonstrate such a transformation, T_{11}, for $\mathbf{P}_{11}$. This transformation forms EDB constants by concatenating the "name" of a gate with a digit 0–9, which designates

the "purpose" of that constant. For example, if e is the name of a gate, constants $e0$ and $e9$, among others, will be created. One "purpose" of these constants is that $p(e0, e9)$ should be true if and only if gate e outputs a 1 in the circuit. In particular, if e is a 1 gate, then T_{11} will put $q_0(e0, e9)$ into the EDB.

First we shall informally explain the principle upon which T_{11} is based. Expand the recursive rule by substituting once for each recursive subgoal. In terms of a derivation, we have

$$P \Rightarrow q_1 P P q_1 \stackrel{*}{\Rightarrow} q_1\ q_1 p p q_1\ q_1 p p q_1\ q_1$$

Then consider an **and** gate e with left input f and right input g, and instantiate the expanded rule as follows (indentation is only to highlight relevant phrases):

$$\begin{aligned} p(e0, e9) :- & \ q_1(e0, e1), \\ & \quad q_1(e1, f0), p(f0, f9), p(f9, f2), q_1(f2, e4), \\ & \quad q_1(e4, g7), p(g7, g0), p(g0, g9), q_1(g9, e8), \\ & q_1(e8, e9). \end{aligned}$$

We next replace p subgoals whose arguments are not of the form $(x0, x9)$ by corresponding q_0 subgoals, and represent the result more succinctly as:

$e0$	$e1$	$f0$	$f9$	$f2$	$e4$	$g7$	$g0$	$g9$	$e8$	$e9$
q_1	q_1	p	q_0	q_1	q_1	q_0	p	q_1	q_1	

We want $p(e0, e9)$ to be derivable if and only if both $p(f0, f9)$ and $p(g0, g9)$ are derivable. T_{11} will put the following atoms into the EDB to make this come about:

- $q_1(e1, f0)$ and $q_1(f2, e4)$ because e has left input f;
- $q_1(e4, g7)$ and $q_1(g9, e8)$ because e has right input g;
- $q_1(e0, e1)$ and $q_1(e8, e9)$ because e has inputs;
- $q_0(f9, f2)$, $q_0(f7, f0)$, $q_0(g9, g2)$, and $q_0(g7, g0)$ because the outputs of f and g are used.

The procedure for an **or** gate e with left input f and right input g is obtained by modifying the **and** procedure. (We eschew a shortcut that is possible in this case, in order to demonstrate a technique that is generally applicable.) Expand the rule as indicated by the derivation:

$$P \Rightarrow q_1 P P q_1 \Rightarrow q_1\ q_1 P P q_1\ P q_1$$

Instantiate the expanded rule twice and replace certain P's by q_0's as before. The first expansion contains constants related to the left input of e:

$e0$	$e1$	$f0$	$f9$	$f2$	$e4$	$e8$	$e9$
q_1	q_1	p	q_0	q_1	q_0	q_1	

and the second expansion contains constants related to the right input of e:

$$\begin{array}{cccccccc} e0 & e1 & g0 & g9 & g2 & e4 & e8 & e9 \\ q_1 & q_1 & p & q_0 & q_1 & q_0 & q_1 & \end{array}$$

We want $p(e0, e9)$ to be derivable if and only if $p(f0, f9)$ or $p(g0, g9)$ (or both) are derivable. T_{11} will put the following atoms into the EDB to make this come about:

- $q_1(e1, f0)$ and $q_1(f2, e4)$ because e has "or" input f;
- $q_1(e1, g0)$ and $q_1(g2, e4)$ because e has "or" input g;
- $q_0(e4, e8)$ because e is an or gate;
- $q_1(e0, e1)$ and $q_1(e8, e9)$ because e has inputs;
- $q_0(f9, f2)$, $q_0(f7, f0)$, $q_0(g9, g2)$, and $q_0(g7, g0)$ because the outputs of f and g are used.

Clearly, T_{11} can run in log space and it produces an EDB that will enable $\mathbf{P}_{11}$ to derive $p(x0, x9)$ if gate x outputs a 1. In the next lemma we show that the "only if" direction also holds.

First we summarize the EDB produced by T_{11} in Fig. 8, according to the gate appearing in the left block of each EDB fact. Let x stand for any gate in the circuit C.

EDB contains:	if x is:
$q_0(x0, x9)$	a 1 gate
$q_0(x4, x8)$	an or gate
$q_0(x7, x0)$	any gate
$q_0(x9, x2)$	any gate
$q_1(x0, x1)$	an and or an or gate
$q_1(x1, f0)$	an and gate with left input f
$q_1(x1, f0)$	an or gate with either input f
$q_1(x2, h4)$	left input to and gate h
$q_1(x2, h4)$	either input to or gate h
$q_1(x4, g7)$	an and gate with right input g
$q_1(x8, x9)$	an and or an or gate
$q_1(x9, h8)$	right input to and gate h

Figure 8: EDB produced by T_{11}

Lemma 9.2: Let C be a circuit in C_2 and let $T_{11}(C)$ be the associated EDB. Let x be any gate in C. We define I_k for all $k \geq 0$ to be the "partial model" consisting of p atoms that are derivable by $\mathbf{P}_{11}$ extended with $T_{11}(C)$ with at most k applications of the recursive rule. Then the following hold for all k, and in particular for the minimum model of $\mathbf{P}_{11} \cup T_{11}(C)$:

(1) Neither $p(x2, Z)$ nor $p(x8, Z)$ is in I_k for any Z.

(2) If $p(x9, Z)$ is in I_k, then $Z = x2$. If $p(x7, Z)$ is in I_k, then $Z = x0$.

(3) If x is an **and** gate with left input f and $p(x1, Z)$ is in I_k, then $Z = e4$ and $p(f0, f9)$ is also in I_k.

(4) If x is an **and** gate with right input g, and $p(x4, Z)$ is in I_k, then $Z = x8$ and $p(g0, g9)$ is also in I_k.

(5) If x is an **or** gate with inputs f and g and $p(x1, Z)$ is in I_k, then $Z = x4$ and $p(f0, f9)$ or $p(g0, g9)$ is also in I_k.

(6) If x is an **and** or an **or** gate and $p(x0, Z)$ is in I_k, then $Z = x9$ and both $p(x1, x4)$ and $p(x4, x8)$ are in I_k.

Proof: We use induction on k, the number of recursive rule applications. Note that nonrecursive rule applications are not counted. Since the minimum model coincides with I_k for some finite k, this will establish the lemma. The basis, $k = 0$, is immediate.

Assume that (1)–(6) hold for I_{k-1}. We shall cite items of this inductive hypothesis as (1H)–(6H). We say that a p atom *reduces to* a chain of subgoals if some substitution unifies the head of the rule with that p atom and unifies the subgoals of the rule to that subgoal chain. Also, we say a reduction *fails on* a subgoal if no instance of that subgoal is in I_{k-1}. Clearly $I_{k-1} \subseteq I_k$.

(1) If $p(x2, Z)$ is derivable in k steps, then it must reduce to

$$q_1(x2, h4), p(h4, V), p(V, W), q_1(W, Z)$$

where h is a gate and $p(h4, V)$ is in I_{k-1}. If h is an **or** gate, no q_1 has $h4$ in its left block, so $q_0(h4, h8)$ is the only possible reduction. If h is an **and** gate, then V must be $h8$ by (4H). Either way, $V = h8$ and the reduction fails on $p(h8, W)$ by (1H). It follows that $p(x2, Z)$ is not in I_k for any Z. Similarly, using (2H), $p(x8, Z)$ must reduce to

$$q_1(x8, x9), p(x9, x2), p(x2, W), q_1(W, Z)$$

which fails on $p(x2, W)$.

(2) The only recursive reduction of $p(x9, Z)$ is

$$q_1(x9, h8), p(h8, V), p(V, W), q_1(W, Z)$$

which fails on $p(h8, V)$, by (1H). There is no recursive reduction at all for $p(x7, Z)$. Therefore the nonrecursive reductions to $q_0(x9, x2)$ and $q_0(x7, x0)$ must be used.

(3) Using (6H) and (2H), $p(x1, Z)$ must reduce to

$$q_1(x1, f0), p(f0, f9), p(f9, f2), q_1(f2, Z)$$

where f is the left input of and gate x. But by the requirements of class C_2, the left output of gate f connects only to x, and the right output of f cannot connect to any gate's left input, so x is the only gate with left input f. Therefore the only instance of $q_1(f2, Z)$ in the EDB is $Z = x4$.

(4) Using (2H) and (6H), $p(x4, Z)$ must reduce to

$$q_1(x4, g7), p(g7, g0), p(g0, g9), q_1(g9, x8)$$

where g is the right input of and gate x. Here we used the fact that x is the only gate with right input g, by membership in C_2.

(5) Using (6H) and (2H), $p(x1, Z)$ must reduce to

$$q_1(x1, f0), p(f0, f9), p(f9, f2), q_1(f2, x4)$$

where f is an input of or gate x.

(6) Using (3H) and (5H), $p(x0, Z)$ must reduce to

$$q_1(x0, x1), p(x1, x4), p(x4, W), q_1(W, Z)$$

Then if x is an and gate, by (4H), $W = x8$. However, if x is an or gate, $p(x4, Z)$ must reduce to $q_0(x4, x8)$, so that again $W = x8$. It follows that $Z = x9$. ∎

Theorem 9.3: $\mathbf{P_{11}}$ is P-complete.

Proof: Reduction from Monotone Circuit Value is accomplished by first transforming the given circuit into an equivalent C_2 circuit, then using T_{11} on that to construct an EDB for $\mathbf{P_{11}}$. The previous lemma shows that the circuit outputs a logical 1 at gate x if and only if $p(x0, x9)$ is a theorem of the resulting extended logic program. ∎

We next consider the elementary single rule program that results if we insert an extra subgoal into $\mathbf{P_4}$, at the left end.

$$\begin{aligned} \mathbf{P_{12}} \qquad & p(X, Y) \text{ :- } p(X, T), p(T, U), q_1(U, V), p(V, W), q_2(W, Y). \\ & p(X, Y) \text{ :- } q_0(X, Y). \end{aligned}$$

which is succinctly expressed by the grammar G_{12} consisting of $P \rightarrow PPq_1Pq_2 \mid q_0$.

Definition 9.1: We say that a recursive subgoal in an elementary chain rule is *completely cut off* if there are nonrecursive goals between it and both ends of the subgoal chain. We say that a recursive subgoal in an elementary chain rule is *partially cut off* if it not at either end of the subgoal chain and the subgoal chain contains at least one nonrecursive subgoal. □

Observe that the subgoal chain in $\mathbf{P}_{12}$ has two occurrences of p that are cut off from the end of the chain, one partially and one completely. The basic idea for the simulation of a circuit is to expand the leftmost p subgoal and assign constants as suggested below.

$$\begin{array}{ccccccccccccccccc} e0 & & f0 & & f9 & & f1 & & f2 & & e4 & & g3 & & g0 & & g9 & & e9 \\ & q_0 & & p & & q_1 & & q_0 & & q_2 & & q_0 & & q_1 & & p & & q_2 & \end{array}$$

Here we assume e is an and-gate with left and right inputs f and g.

EDB contains:	if x is:
$q_0(x0, x9)$	a **1** gate
$q_0(x0, f0)$	an **and** gate with left input f
$q_0(x0, f0)$	an **or** gate with either input f
$q_0(x1, x2)$	any gate
$q_0(x4, g3)$	an **and** gate with right input g
$q_0(x4, x5)$	an **or** gate
$q_0(x6, x7)$	an **or** gate
$q_1(x3, x0)$	any gate
$q_1(x5, x6)$	an **or** gate
$q_1(x9, x1)$	any gate
$q_1(x0, x1)$	an **and** or an **or** gate
$q_2(x2, h4)$	left input to **and** gate h
$q_2(x2, h4)$	either input to **or** gate h
$q_2(x7, x9)$	an **or** gate
$q_2(x9, h9)$	right input to **and** gate h

Figure 9: EDB produced by T_{12}, where x is any gate.

Theorem 9.4: $\mathbf{P}_{12}$ is P-complete.

Proof: Define transformation T_{12} to transform a circuit $C \in C_2$ into the EDB described by Fig. 9. The inductive hypothesis is:

- $p(e0, Z)$ is derivable $\Rightarrow$ $Z = f0$ or $Z = e4$ or $Z = e9$. If $Z = e4$, then $p(f0, f9)$ is derivable, where f is the left input of **and**-gate e. If $Z = e9$, then $p(e0, e4)$ is derivable, and $p(g0, g9)$ is derivable, where g is the right input of **and**-gate e. (There are obvious adjustments when e is an **or**-gate.)
- $p(x1, Z)$ is derivable $\Rightarrow$ $Z = x2$.
- $p(e4, Z)$ is derivable $\Rightarrow$ $Z = g3$, where g is the right input of **and**-gate e. (Adjust for **or**-gate.)
- No instances of $p(x2, Z)$, $p(x3, Z)$, or $p(x9, Z)$ are derivable.

The argument follows the same lines as for $\mathbf{P}_{11}$, so further details are omitted. ∎

Conjecture Any elementary single rule program in which the recursive rule has two or more cut off recursive subgoals, at least one of them completely cut off, is $\mathcal{P}$-complete.

The ideas used on $\mathbf{P}_{11}$ and $\mathbf{P}_{12}$ appear to be generalizable. However, a word of caution is in order. If a certain template, such as $\mathbf{P}_{11}$, is $\mathcal{P}$-complete, it is tempting to conclude that more complicated programs, in which $\mathbf{P}_{11}$ can be embedded, are also $\mathcal{P}$-complete. However, this is not automatically true; it needs to be proved.

9.2 Other $\mathcal{P}$-Complete Single Rule Programs

In this section we briefly summarize a *pot pourri* of $\mathcal{P}$-completeness results for various logical query programs.

Path Systems [Coo74,JL76] is well known to be $\mathcal{P}$-complete, and is in fact the "first" $\mathcal{P}$-complete problem. It is very naturally represented as the logical query program

$$\mathbf{P}_{21} \qquad \begin{array}{l} r(X) :\!- h(X, Y, Z), r(Y), r(Z). \\ r(X) :\!- s(X). \end{array}$$

where s and h are EDB relations specifying "sources" and "hyperedges," and r signifies "reachable."

Example 9.1: Cosmadakis and Kanellakis [CK85] discovered that an interesting variant of *Path Systems*, called the *blue-blooded Frenchman* problem, is also $\mathcal{P}$-complete:

$$\mathbf{P}_{22} \qquad \begin{array}{l} r(X) :\!- h_1(X, Y), r(Y), h_2(X, Z), r(Z). \\ r(X) :\!- s(X). \end{array}$$

Essentially this means that we can require h to be the equi-join of h_1 and h_2 in *Path Systems*. An easy proof for $\mathbf{P}_{22}$ is by reduction from Monotone Circuit Value. Let C be a given circuit in $\mathcal{C}_1$.

- For each **and** gate e with inputs f and g we put $h_1(e, f)$ and $h_2(e, g)$ in the EDB to signify that e has f as its left input and g as its right input.

- For each **or** gate e with inputs f and g we put $h_1(e,f)$ and $h_1(e,g)$ in the EDB to signify that e has f and g as its inputs, and we put in $h_2(e,e')$ and $s(e')$ to allow derivations to be completed.
- For each **1** gate e we put $s(e)$ in the EDB.

An easy induction on proof length shows that if $r(X)$ is true, then either $X = e'$ for some **or** gate $e \in C$, or $X = e$ for some gate $e \in C$ that outputs a 1.

A more intriguing approach is to simulate a *scanning Turing machine*[6] directly with $\mathbf{P}_{22}$. The main idea is use h_1 to verify the tape symbol and use h_2 to verify the state transition in such a way that $r(X)$ is true exactly when X is a reachable *local ID*, consisting of an encoding of (*time, head-position, state, symbol-scanned*). The details are omitted, as the ideas can be found in [Coo74], where it is shown that any language in P can be accepted by some scanning Turing machine. □

Let us define *pure recursion* as the use of recursive rules in which every subgoal is recursive. For example, $\mathbf{P}_1$ and $\mathbf{P}_2$ employ pure recursion. The next example demonstrates that pure recursion can be P-complete.

Example 9.2: The following program employs pure recursion and has a single recursive rule in which the subgoals represent a chain, but not an elementary chain. That is, the left block and right block are defined for each literal, but different literals with the same predicate symbol have different block structures.

$$\mathbf{P}_{23} \qquad \begin{array}{l} p(W,X,Y,Z) \text{ :- } p(U,W,U,V), p(V,X,Y,Z). \\ p(W,X,Y,Z) \text{ :- } q_0(W,X,Y,Z). \end{array}$$

For the rule head and the rightmost subgoal, the left block is argument 1 and the right block is arguments 2, 3, and 4. But for the left subgoal, the left block is argument 2 and the right block is argument 4; arguments 1 and 3 are in neither block.

Proof of P-completeness is by reduction from Path Systems. Given an EDB for $\mathbf{P}_{21}$, we encode q_0 as the cross product, $s \times h$. It then follows that the minimum model of $\mathbf{P}_{23}$ will be the cross product $p = r \times h$, where r is the minimum model of $\mathbf{P}_{21}$. □

Our final example has the interesting property that it is P-complete for unrestricted EDBs, yet is in NC for EDBs in which the q_0 relation is "acyclic," in the sense defined in the example.

Example 9.3: Consider the following program, called *Transitive Closure with Permissions.* It exhibits a nonchain rule:

$$\mathbf{P}_{24} \qquad \begin{array}{l} p(X,Y) \text{ :- } q_1(X,Y), p(X,U), p(U,Y). \\ p(X,Y) \text{ :- } q_0(X,Y). \end{array}$$

[6] A scanning TM scans the tape in the fixed sequence 0, 1, 0, -1, 0, 1, 2, 1, 0, -1, -2, ...

The proof of P-completeness for unrestricted EDBs is by reduction from Monotone Circuit Value for circuits in C_1. To sketch the main idea, we want **and** gate e with inputs f and g to output a 1 if and only if $p(e0, e9)$ is derivable, and we want $p(e0, e9)$ to be derivable only if both $p(e0, e4)$ and $p(e4, e9)$ are derivable. Thus we put $q_1(e0, e9)$ in the EDB. By judicious choice of q_1 and q_0 EDB facts we ensure that

- $p(e0, e4)$ is derivable if and only if $p(f0, f9)$ is derivable;
- $p(e4, e9)$ is derivable if and only if $p(g0, g9)$ is derivable.

Additional details may be worked out by the interested reader.

It is also easy to see that $\mathbf{P_{24}}$ is in NC when the EDB is restricted to include any q_1 relation, but only q_0 relations that define the edges of a directed acyclic graph. For the q_0 atoms in the fringe of any complete derivation tree form a chain, and constitute a constant fraction of all atoms in the fringe. Thus $\mathbf{P_{24}}$ has the Polynomial Fringe Property when restricted to this class of EDBs. □

10 Conclusion and Open Problems

We have defined the Polynomial Fringe Property for logical query programs and shown that programs with this property are in NC. For programs with elementary chain rules, the GSM Mapping Theorem gives a sufficient condition that they have the Polynomial Fringe Property, in terms of GSM mappings of $D_1\$$, the Dyck language on one kind of parentheses, with an end-marker appended.

Our applications of the GSM Mapping Theorem (see $\mathbf{P_4}$, $\mathbf{P_5}$, $\mathbf{P_7}$) were rather *ad hoc*: We first characterized the language, then found a GSM to generate it, and a deterministic one at that. Is there some interesting class of grammars for which we can go directly to GSM's? Can we use non-determinism?

In Section 9, we conjectured that two cut off recursive subgoals in an elementary single rule program ensures P-completeness. Is this conjecture true?

Bottom-up evaluation of the entire minimum model of a logical query program is not usually considered practical. The application of the techniques in this paper to a mixed bottom-up and top-down strategy that allows a high degree of parallelism, yet only works on "somewhat relevant" portions of the minimum model, represents a significant open problem.

References

[AH85] Mikhail J. Atallah and Susanne E. Hambrusch. Solving tree problems on a mesh-connected processor array. In *26th Symposium on Foundations of Computer Science*, pages 222–231, 1985.

[AU79] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *6th ACM Symp. on Principles of Programming Languages*, pages 110–117, 1979.

[AVE82] K. R. Apt. and M. H. Van Emden. Contributions to the theory of logic programming. *JACM*, 29(3):841–862, 1982.

[CH82] Ashok Chandra and David Harel. Structure and complexity of relational queries. *JCSS*, 25(1):99–128, 1982.

[CK85] Stavros S. Cosmadakis and Paris C. Kanellakis. *Parallel evaluation of recursive rule queries.* To appear in *5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Mar. 1986.

[Coo74] Stephen A. Cook. An observation on time-storage trade-off. *JCSS*, 9(3):308–316, 1974.

[Coo84] Stephen A. Cook. *A taxonomy of problems with fast parallel algorithms.* Technical Report 164/83, Dept. of Computer Science, University of Toronto, Oct. 1984.

[Gre73] Sheila A. Greibach. The hardest context-free language. *SIAM J. Computing*, 2(4):304–310, 1973.

[GS66] S. Ginsburg and E. H. Spanier. Finite turn pushdown automata. *SIAM J. Control*, 4(3):429–453, 1966.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, Reading, MA, 1979.

[JL76] Neil D. Jones and William T. Laaser. Complete problems for deterministic polynomial time. *Theoretical Computer Science*, 3(1):105–117, Oct. 1976.

[MR85] Gary L. Miller and John H. Reif. Parallel tree contraction and its applications. In *26th Symposium on Foundations of Computer Science*, pages 478–489, 1985.

[Ruz80] Walter L. Ruzzo. Tree-size bounded alternation. *JCSS*, 21(2):218–235, Oct. 1980.

[Var82] Moshe Y. Vardi. Complexity of relational queries. In *14th ACM Symposium on Theory of Computing*, pages 137–145, 1982.

[Var85] Moshe Y. Vardi. Querying logical databases. In *4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 57–65, 1985.

[VEK76] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *JACM*, 23(4):733–742, 1976.

[VSBR83] L. G. Valiant, S. Skyum, S. Berkowitz, and C. Rackoff. Fast parallel computation on polynomials using few processors. *SIAM J. Computing*, 12(4):641–644, Nov. 1983.

END

FILMED

6-86

DTIC